Embedded Systems Engineering
- Gesamtausgabe -

V 0.15

Prof. Dr. Christian Siemers

TU Clausthal
FH Nordhausen
Private FH Göttingen

Inhaltsverzeichnis

Abschnitt I: Design von Eingebetteten Systemen

1 Einführung in informationstechnische Systeme und Informationsverarbeitung

Eingebettete Systeme (*embedded systems*) sind Computersysteme, die aus Hardware und Software bestehen und die in komplexe technische Umgebungen eingebettet sind [Sch05]. Diese Umgebungen sind meist maschinelle Systeme, in denen das eingebettete System mit Interaktion durch einen Benutzer arbeitet oder auch vollautomatisch (autonom) agiert. Die eingebetteten Systeme übernehmen komplexe Steuerungs-, Regelungs- und Datenverarbeitungsaufgaben für bzw. in diesen technischen Systemen.

Der Begriff *Informationstechnische Systeme* umfasst wesentlich mehr als nur die eingebetteten Systeme, nämlich grundsätzlich alle Rechnersysteme. Hierunter wird im Allgemeinen ein System verstanden, das in der Lage ist, ein (berechenbares) Problem zu berechnen bzw. zu lösen. Wie dieses Problem dem informationstechnischen System zugänglich gemacht wird, bleibt zunächst offen, doch existieren hierfür zwei Wege: Herstellung eines spezifischen IT-Systems oder Verwendung eines allgemeinen IT-Systems („universelle Maschine") und Verfassen eines Programms (= außerhalb der Herstellungsprozesses hergestellte und dem System zugänglich gemachte algorithmische Beschreibung) für das spezifische Problem.

Diese Vorlesung beschränkt sich auf die *programmierbaren* informationstechnischen Systeme und darin speziell denjenigen, die als eingebettete Systeme eingesetzt werden. Dies ist keine wirkliche Beschränkung, sondern eher eine Fokussierung, da gerade das Design dieser Systeme eher schwieriger zu bewerten ist als das allgemeiner Rechner und deren Anwendungen.

Die Vorlesung wurde weiterhin so konzipiert, dass die Software im Vordergrund steht. Es geht um (binärwertige) digitale Systeme, die einerseits programmierbar sind, andererseits bei der Ausführung einen Zeitverlauf aufweisen, und deren Entwurf insbesondere in eingebetteten Systemen. Hierzu sollte gleich zu Beginn beachtet werden, dass mit System sowohl das Rechnersystem als auch die relevante Umgebung bezeichnet sein kann. Um hier Verwirrungen zu vermeiden, sei für diesen Kurs mit *System* das digitale System gemeint, also dasjenige, das konzipiert und konstruiert werden soll, während die Umgebung mit *Prozess* oder – präziser – mit *Umgebungsprozess* bezeichnet wird.

Im Vordergrund steht also das System. Die eingebetteten Systeme zeigen dabei eine große Spannweite, denn es ist ein großer Unterschied, eine Kaffeemaschine oder ein Flugzeug zu steuern. Zunächst muss also einmal klassifiziert werden, um die Vielfalt zu beherrschen, und dann werden bestimmte Teile näher behandelt.

Im Anschluss daran soll verdeutlicht werden, worin die eigentlichen Schwierigkeiten bei der Entwurfsmethodik bestehen werden: Der Umgebungsprozess setzt Randbedingungen, und diese Randbedingungen (constraints) müssen neben der

algorithmischen Richtigkeit zusätzlich eingehalten werden. Dies wird anhand der Zeitbedingungen deutlich werden (Abschnitt 1.3).

1.1 Klassifizierung

Definition 1.1:

Ein *eingebettetes System (embedded system)* ist ein binärwertiges digitales System (Computersystem), das in ein umgebendes technisches System eingebettet ist und mit diesem in Wechselwirkung steht.

Das Gegenstück zu *Embedded System* wird *Self-Contained System* genannt. Als Beispiele können Mikrocontroller-basierte Systeme im Auto, die Computertastatur usw. genannt werden.

Hinweis: Die Definition der eingebetteten Systeme ist eine ”weiche“ Definition, aber sie ist trotzdem sehr wichtig! Der Grund bzw. der Unterschied zu den Self-Contained Rechnern besteht darin, dass – wie erwähnt – die Korrektheit bzw. Erfüllung auch in den Randbedingungen (und nicht nur im Algorithmus) einzuhalten ist.

1.1.1 Allgemeine Klassifizierung von Computersystemen

Die heute verfügbaren Computersysteme können in drei unterschiedliche Klassen eingeteilt werden [Sch05]: (rein) transformationelle, interaktive und reaktive Systeme. Die Unterscheidung erfolgt in erster Linie durch die Art und Weise, wie Eingaben in Ausgaben transformiert werden.

Transformationelle Systeme transformieren nur solche Eingaben in Ausgaben, die zum Beginn der Systemverarbeitung vollständig vorliegen [Sch05]. Die Ausgaben sind nicht verfügbar, bevor die Verarbeitung terminiert. Dies bedeutet auch, dass der Benutzer bzw. die Prozessumgebung nicht in der Lage ist, während der Verarbeitung mit dem System zu interagieren und so Einfluss zu nehmen.

Interaktive Systeme erzeugen Ausgaben nicht nur erst dann, wenn sie terminieren, sondern sie interagieren und synchronisieren stetig mit ihrer Umgebung [Sch05]. Wichtig hierbei ist, dass diese Interaktion durch das Rechnersystem bestimmt wird, nicht etwa durch die Prozessumgebung: Wann immer das System neue Eingaben zur Fortführung benötigt, wird die Umgebung, also ggf. auch der Benutzer hierzu aufgefordert. Das System synchronisiert sich auf diese *proaktive* Weise mit der Umgebung.

Bei *reaktiven Systemen* schreibt die Umgebung vor, was zu tun ist [Sch05]. Das Computersystem reagiert nur noch auf die externen Stimuli, die Prozessumgebung synchronisiert den Rechner (und nicht umgekehrt).

Worin liegen die Auswirkungen dieses kleinen Unterschieds, wer wen synchronisiert? Die wesentlichen Aufgaben eines interaktiven Systems sind die Vermeidung von Verklemmungen (deadlocks), die Herstellung von "Fairness" und die Erzeugung einer Konsistenz, insbesondere bei verteilten Systemen. Reaktive Systeme hingegen verlangen vom Computer, dass dieser reagiert, und zwar meistens rechtzeitig. Rechtzeitigkeit und Sicherheit sind die größten Belange dieser Systeme.

Zudem muss von interaktiven Systemen kein deterministisches Verhalten verlangt werden: Diese können intern die Entscheidung darüber treffen, wer wann bedient wird. Selbst die Reaktion auf eine Sequenz von Anfragen muss nicht immer gleich sein. Bei reaktiven Systemen ist hingegen der Verhaltensdeterminismus integraler Bestandteil. Daher hier die Definition von Determinismus bzw. eines deterministischen Systems:

Definition 1.2:

Ein System weist *determiniertes* oder *deterministisches Verhalten* (*Deterministic Behaviour*) auf, wenn zu jedem Satz von inneren Zuständen und jedem Satz von Eingangsgrößen genau ein Satz von Ausgangsgrößen gehört.

Als Gegenbegriffe können stochastisch oder nicht-deterministisch genannt werden. Diese Definition bezieht sich ausschließlich auf die logische (algorithmische) Arbeitsweise, und das klassische Beispiel sind die endlichen Automaten (DFA, Deterministic Finite Automaton). Nicht-deterministische Maschinen werden auf dieser Ebene in der Praxis nicht gebaut, beim NFA (Non-Deterministic Finite Automaton) handelt es sich um eine theoretische Maschine aus dem Gebiet der Theoretischen Informatik.

1.1.2 Klassifizierung eingebetteter Systeme

Eingebettete Systeme, die mit einer Umgebung in Wechselwirkung stehen, sind nahezu immer als reaktives System ausgebildet. Interaktive Systeme sind zwar prinzipiell möglich, doch die Einbettung macht in der Regel eine Reaktivität notwendig. Die wichtigsten Eigenschaften im Sinn der Einbettung sind: Nebenläufigkeit (zumindest oftmals), hohe Zuverlässigkeit und Einhaltung von Zeitschranken.

Noch eine Anmerkung zum Determinismus: Während man davon ausgehen kann, dass alle technisch eingesetzten, eingebetteten Systeme deterministisch sind, muss dies für die Spezifikation nicht gelten: Hier sind nicht-deterministische Beschreibungen erlaubt, z.B., um Teile noch offen zu lassen.

Wird die Einhaltung von Zeitschranken zu einer Hauptsache, d.h. wird die Verletzung bestimmter Zeitschranken sehr kritisch im Sinn einer Gefährdung für Mensch und Maschine, dann spricht man von Echtzeitsystemen. Echtzeitfähige eingebettete Systeme sind eine echte Untermenge der reaktiven Systeme, die ihrerseits eine echte Untermenge der eingebetteten Systeme darstellen (Bild 1.1).

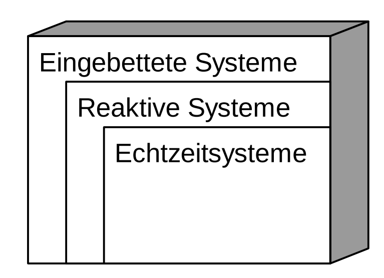


Bild 1.1 Klassifikation eingebetteter Systeme

Eingebettete Systeme lassen sich weiterhin nach einer Reihe von unterschiedlichen Kriterien klassifizieren. Hierzu zählen:

- *Kontinuierlich* versus *diskret*: Diese Ausprägung der Stetigkeit bezieht sich sowohl auf Datenwerte als auch auf die Zeit (→ 1.2). Enthält ein System beide Verhaltensweisen, wird es als ”hybrides System“ bezeichnet.
- *Monolithisch* versus *verteilt*: Während anfänglich alle Applikationen für eingebettete Systeme als monolithische Systeme aufgebaut wurden, verlagert sich dies zunehmend in Richtung verteilte Systeme. Hier sind besondere Anforderungen zu erfüllen, wenn es um Echtzeitfähigkeit geht.
- *Sicherheitskritisch* versus *nicht-sicherheitskritisch*: Sicherheitskritische Systeme sind solche, deren Versagen zu einer Gefährdung von Menschen und/oder Einrichtungen führen kann. Viele Konsumprodukte sind sicherheits-unkritisch, während Medizintechnik, Flugzeugbau sowie Automobile zunehmend auf sicherheitskritischen eingebetteten Systemen beruhen.

1.1.3 Definitionen

In diesem Abschnitt werden einige Definitionen gegeben, die u.a. [Sch05] entnommen sind. Diese Definitionen beziehen sich im ersten Teil auf die informationstechnische Seite, weniger auf die physikalisch-technische.

Definition 1.3:

Unter einem *System* versteht man ein mathematisches Modell **S**, das einem Eingangssignal der Größe x ein Ausgangssignal y der Größe $y = \mathbf{S}(x)$ zuordnet.

Wenn das Ausgangssignal hierbei nur vom aktuellen Wert des Eingangssignals abhängt, spricht man von einem *gedächtnislosen* System (Beispiel: Schaltnetze in der digitalen Elektronik). Hängt dagegen dieser von vorhergehenden Eingangssignalen ab, spricht man von einem *dynamischen* System (Beispiel: Schaltwerke).

Definition 1.4:

Ein *reaktives System* (*reactive system*) kann aus Software und/oder Hardware bestehen und setzt Eingabeereignisse, deren zeitliches Verhalten meist nicht vor-

hergesagt werden kann, in Ausgabeereignisse um. Die Umsetzung erfolgt oftmals, aber nicht notwendigerweise unter Einhaltung von Zeitvorgaben.

Definition 1.5:

Ein *hybrides System* (*hybrid system*) ist ein System, das sowohl kontinuierliche (analoge) als auch diskrete Datenanteile (wertkontinuierlich) verarbeiten und/oder sowohl über kontinuierliche Zeiträume (zeitkontinuierlich) als auch zu diskreten Zeitpunkten mit ihrer Umgebung interagieren kann.

Definition 1.6:

Ein *verteiltes System* (*distributed system*) besteht aus Komponenten, die räumlich oder logisch verteilt sind und mittels einer Kopplung bzw. Vernetzung zum Erreichen der Funktionalität des Gesamtsystems beitragen. Die Kopplung bzw. Vernetzung spielt bei echtzeitfähigen Systemen eine besondere Herausforderung dar.

Definition 1.7:

Ein *Steuergerät* (*electronic control unit*, ECU) ist die physikalische Umsetzung eines eingebetteten Systems. Es stellt damit die Kontrolleinheit eines mechatronischen Systems dar. In mechatronischen Systemen bilden Steuergerät und Sensorik/Aktorik oftmals eine Einheit.

Defnition 1.8:

Wird Elektronik zur Steuerung und Regelung mechanischer Vorgänge räumlich eng mit den mechanischen Systembestandteilen verbunden, so spricht man von einem *mechatronischen System*. Der Forschungszweig, der sich mit den Grundlagen und der Entwicklung mechatronische Systeme befasst, heißt *Mechatronik* (*mechatronics*).

Mechatronik ist ein Kunstwort, gebildet aus *Mecha*nik und Elek*tronik*. In der Praxis gehört allerdings eine erhebliche Informatik-Komponente hinzu, da nahezu alle mechatronischen Systeme auf Mikrocontrollern/Software basieren

1.2 Aufbau und Komponenten eingebetteter Systeme

Während der logische Aufbau eingebetteter Systeme oftmals sehr ähnlich ist – siehe unten – hängt die tatsächliche Realisierung insbesondere der Hardware stark von den Gegebenheiten am Einsatzort ab. Hier können viele Störfaktoren herrschen, zudem muss das eingebettete System Sorge dafür tragen, nicht selbst zum Störfaktor zu werden.

Einige *Störfaktoren* sind: Wärme/Kälte, Staub, Feuchtigkeit, Spritzwasser, mechanische Belastung (Schwingungen, Stöße), Fremdkörper, elektromagnetische Störungen und Elementarteilchen (z.B. Höhenstrahlung). Allgemeine und Herstellerspezifische Vorschriften enthalten teilweise genaue Angaben zur Vermeidung des

passiven und aktiven Einflusses, insbesondere im EMV-Umfeld (Elektromagnetische Verträglichkeit). Dieses Gebiet ist nicht Bestandteil dieser Vorlesung, aber es soll an dieser Stelle darauf hingewiesen werden.

Der *logische Aufbau* der eingebetteten Systeme ist jedoch recht einheitlich, in der Regel können 5 strukturelle Bestandteile identifiziert werden [Sch05]:

- Die Kontrolleinheit bzw. das Steuergerät (→ Definition 1.7), d.h. das eingebettete Hardware/Software System,
- die Regelstrecke mit Aktoren (bzw. Aktuatoren) (actuator) und Sensoren (sensor), d.h. das gesteuerte/geregelte physikalische System,
- die Benutzerschnittstelle,
- die Umgebung sowie
- den Benutzer.

Mit stark zunehmender Tendenz werden diese Systeme noch vernetzt, so dass sich neben der lokalen Ebene noch eine globale Vernetzungsebene mit physikalischem Zugang zur Kontrolleinheit und logischem Zugang zu allen Komponenten des Systems ergibt.

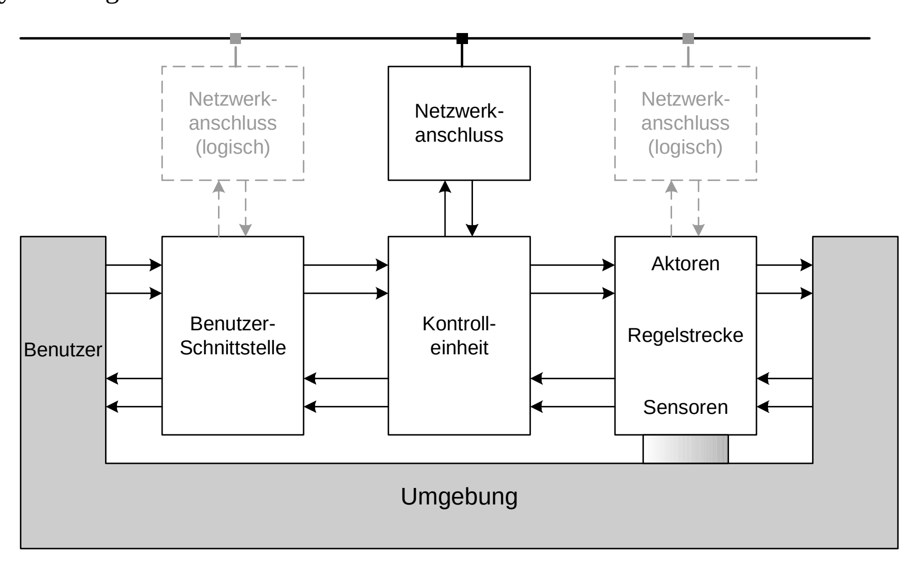


Bild 1.2 Erweiterte Referenzarchitektur eines eingebetteten Systems [Sch05]

Bild 1.2 stellt diese Referenzarchitektur eines eingebetteten Systems als Datenflussarchitektur dar, in der die Pfeile die gerichteten Kommunikationskanäle zeigen. Solche Kommunikationskanäle können (zeit- und wert-)kontinuierliche Signale oder Ströme diskreter Nachrichten übermitteln. Regelstrecke und Umgebung

sind hierbei auf meist komplexe Weise miteinander gekoppelt, die schwer formalisierbar sein kann.

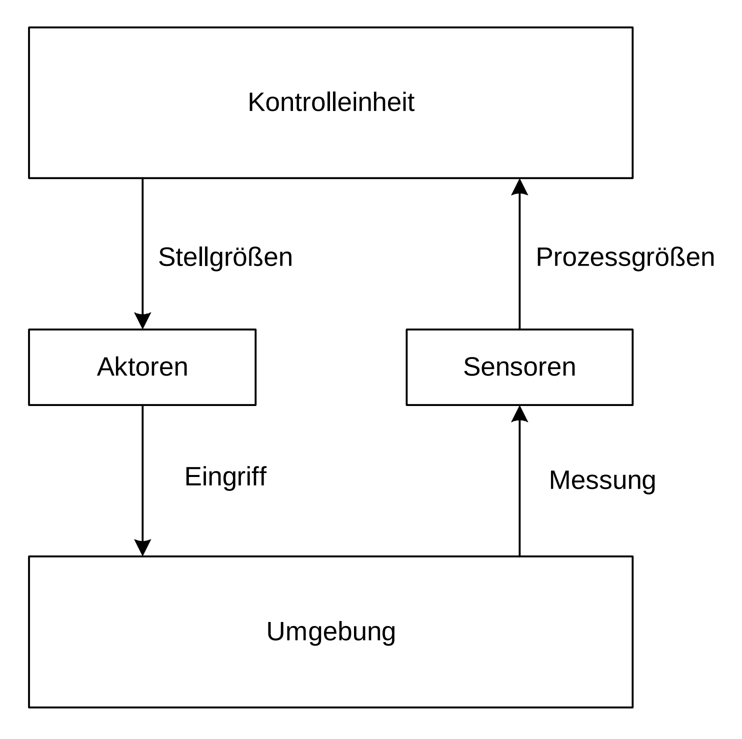


Bild 1.3 Wirkungskette System/Umgebung

Bild 1.3 zeigt die geschlossene Wirkungskette, die ein eingebettetes System einschließlich der Umgebung bildet. Der zu regelnde oder steuernde Prozess ist über Sensoren und Aktoren an das Steuergerät gekoppelt und kommuniziert mit diesem darüber. Sensoren und Aktoren fasst man unter dem (aus dem Von-Neumann-Modell wohlbekannten) Begriff Peripherie (peripheral devices) oder I/O-System (input/output) zusammen.

Zu den einzelnen Einheiten seien einige Anmerkungen hier eingeführt:

Kontrolleinheit

Die Kontrolleinheit bildet den Kern des eingebetteten Systems, wobei sie selbst wieder aus verschiedenen Einheiten zusammengesetzt sein kann. Sie muss das Interface zum Benutzer (falls vorhanden) und zur Umgebung bilden, d.h., sie empfängt Nachrichten bzw. Signale von diesen und muss sie in eine Reaktion umsetzen.

Wie bereits in Abschnitt 1.1.2 dargestellt wurde ist diese Kontrolleinheit fast ausschließlich als reaktives System ausgeführt. Die Implementierung liegt in modernen Systemen ebenso fast ausnahmslos in Form programmierbarer Systeme, also als Kombination Hardware und Software vor. Hierbei allerdings gibt es eine Vielzahl von Möglichkeiten: ASIC (Application-Specific Integrated Circuit), PLD/ FPGA (Programmable Logic Devices/Field-Programmable Gate Arrays), General-Purpose Mikrocontroller, DSP (Digital Signal Processor), ASIP (Application-Specific Instruction Set Processor), um nur die wichtigsten Implementierungsklassen zu nennen. Man spricht hierbei von einem Design Space bzw. von Design Space Exploration (→ 5).

Peripherie: Analog/Digital-Wandler

Ein Analog/Digital-Wandler (Analog/Digital-Converter, ADC), kurz A/D-Wandler, erzeugt aus einem (wert- und zeit-)analogen Signal digitale Signale. Die Umsetzung ist ein vergleichsweise komplexer Prozess, der in Bild 1.4 dargestellt ist. Hierbei handelt es sich nicht um eine Codierung, und der Prozess ist nicht exakt reversibel.

Der technisch eingeschlagene Weg besteht aus der Abtastung zuerst (Bauteil: Sample&Hold- bzw. Track&Hold-Schaltung), gefolgt von einer Quantisierung und der Codierung. Die Abtastung ergibt die Zeitdiskretisierung, die Quantisierung die Wertediskretisierung. Man beachte, dass mit technischen Mitteln sowohl die Abtastfrequenz als auch die Auflösung zwar ”beliebig“ verbessert werden kann, aber niemals kontinuierliche Werte erreicht werden. In eingebetteten Systemen werden diese Werte den Erfordernissen der Applikation angepasst.

Für die Umsetzung von analogen Werten in digiatle Werte sind verschiedene Verfahren bekannt: Flash, Half-Flash, Semi-Flash, Sukzessive Approximation, Sigma-Delta-Wandler usw.

Peripherie: Digital/Analog-Wandler

Der Digital/Analog-Wandler, kurz D/A-Wandler (Digital/Analog-Converter, DAC) erzeugt aus digitalen Signalen ein analoges Signal (meist eine Spannung). Dies stellt die Umkehrung der A/D-Wandlung dar. Die Umsetzung erfolgt exakt, abgesehen von Schaltungsfehlern, d.h. ohne prinzipiellen Fehler wie bei der A/D-Wandlung.

Gängige Verfahren sind: Pulsweiten-Modulation (pulse width modulation, PWM) und R-2R-Netzwerke.

Peripherie: Sensoren

Zunächst sei die Definition eines Sensors gegeben [Sch05]:

Definition 1.9:

Ein *Sensor* ist eine Einrichtung zum Feststellen von physikalischen oder chemischen Eingangsgrößen, die optional eine Messwertzuordnung (Skalierung) der Größen treffen kann, sowie ggf. ein digitales bzw. digitalisierbares Ausgangssignal liefern kann.

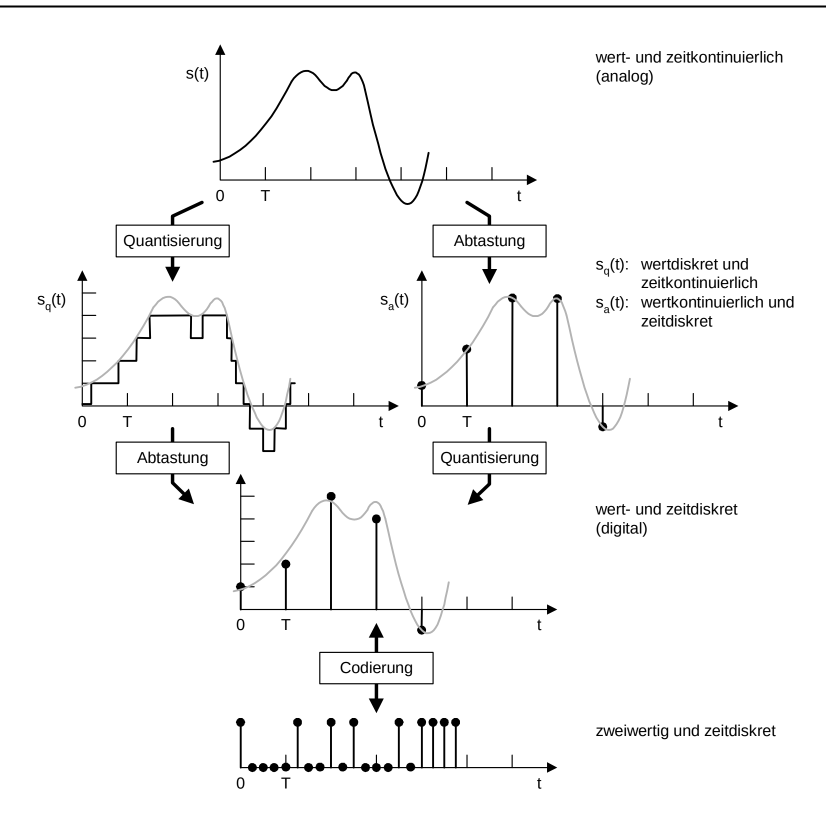


Bild 1.4 Vorgänge bei der AD-Wandlung

Sensoren stellen also das primäre Element in einer Messkette dar und setzen variable, im Allgemeinen nichtelektrische Eingangsgrößen in ein geeignetes, insbesondere elektrisches Messsignal um. Hierbei können ferner *rezeptive Sensoren*, die nur passiv Signale umsetzen (Beispiel: Mikrofon), sowie *signalbearbeitende Sensoren*, die die Umwelt stimulieren und die Antwort aufnehmen (Beispiel: Ultraschall-Sensoren zur Entfernungsmessung), unterschieden werden.

Als *Smart Sensors* bezeichnete Sensoren beinhalten bereits eine Vorverarbeitung der Daten. Hierdurch sind Netzwerke von Sensoren möglich, die auch ganz neue Strategien wie gegenseitige Überwachungen bzw. Plausibilitätskontrollen ermöglichen.

Peripherie: Aktuatoren

Aktuatoren bzw. Aktoren verbinden den informationsverarbeitenden Teil eines eingebetteten Systems und den Prozess. Sie wandeln Energie z.B. in mechanischen Arbeit um.

Die Ansteuerung der Aktuatoren kann analog (Beispiel: Elektromotor) oder auch digital (Beispiel: Schrittmotor) erfolgen.

1.3 Die Rolle der Zeit und weitere Randbedingungen

1.3.1 Verschiedene Ausprägungen der Zeit

In den vorangegangenen Abschnitten wurde bereits verdeutlicht: Die Zeit spielt bei den binärwertigen digitalen *und* den analogen Systemen (Umgebungsprozess) eine Rolle, die genauer betrachtet werden muss. Wir unterscheiden folgende Zeitsysteme:

Definition 1.10:

In *Zeit-analogen Systemen* ist die Zeit komplett kontinuierlich, d.h., jeder Zwischenwert zwischen zwei Zeitpunkten kann angenommen werden und ist Werterelevant.

Als Folge hiervon muss jede Funktion f(t) für alle Werte $t \in [-\infty, \infty]$ bzw. für endliche Intervalle mit $t \in [t_0, t_1]$ definiert werden. Zeit-analoge Systeme sind fast immer mit Werte-Analogie gekoppelt. Zusammengefasst wird dies als analoge Welt bezeichnet.

Definition 1.11:

In *Zeit-diskreten Systemen* gilt, dass das System, beschrieben z.B. durch eine Funktion, von abzählbare vielen Zeitpunkte abhängt. Hierbei können abzählbar unendlich viele oder endlich viele Zeitpunkte relevant sein.

Folglich wird jede Funktion g(t) für alle Werte $t \in N$ (oder ähnlich mächtige Mengen) oder für $t \in \{t_0, t_1, \ldots t_k\}$ definiert. Zeit-diskrete Systeme sind fast immer mit Werte-Diskretheit gekoppelt, man spricht dann auch von der digitalen Welt.

Definition 1.12:

Zeit-unabhängige Systeme sind Systeme, die keine Zeitbindung besitzen. Dies bedeutet nicht, dass sie über die Zeit konstant sind, sie sind nur nicht explizit daran gebunden.

Hiermit wird deutlich, dass zwischen einer realen Zeit (Außenzeit) und der Programmlaufzeit (Innenzeit) unterschieden werden muss. Die Aufgabe einer Echtzeitprogrammierung besteht also darin, zwischen realer Zeit und Programmlaufzeit

eine feste Beziehung herzustellen. Die Zeit-unabhängigen Systeme werden häufig auch als informationstechnische Systeme (IT-Systeme) bezeichnet (siehe hierzu auch die Einleitung zu diesem Kapitel).

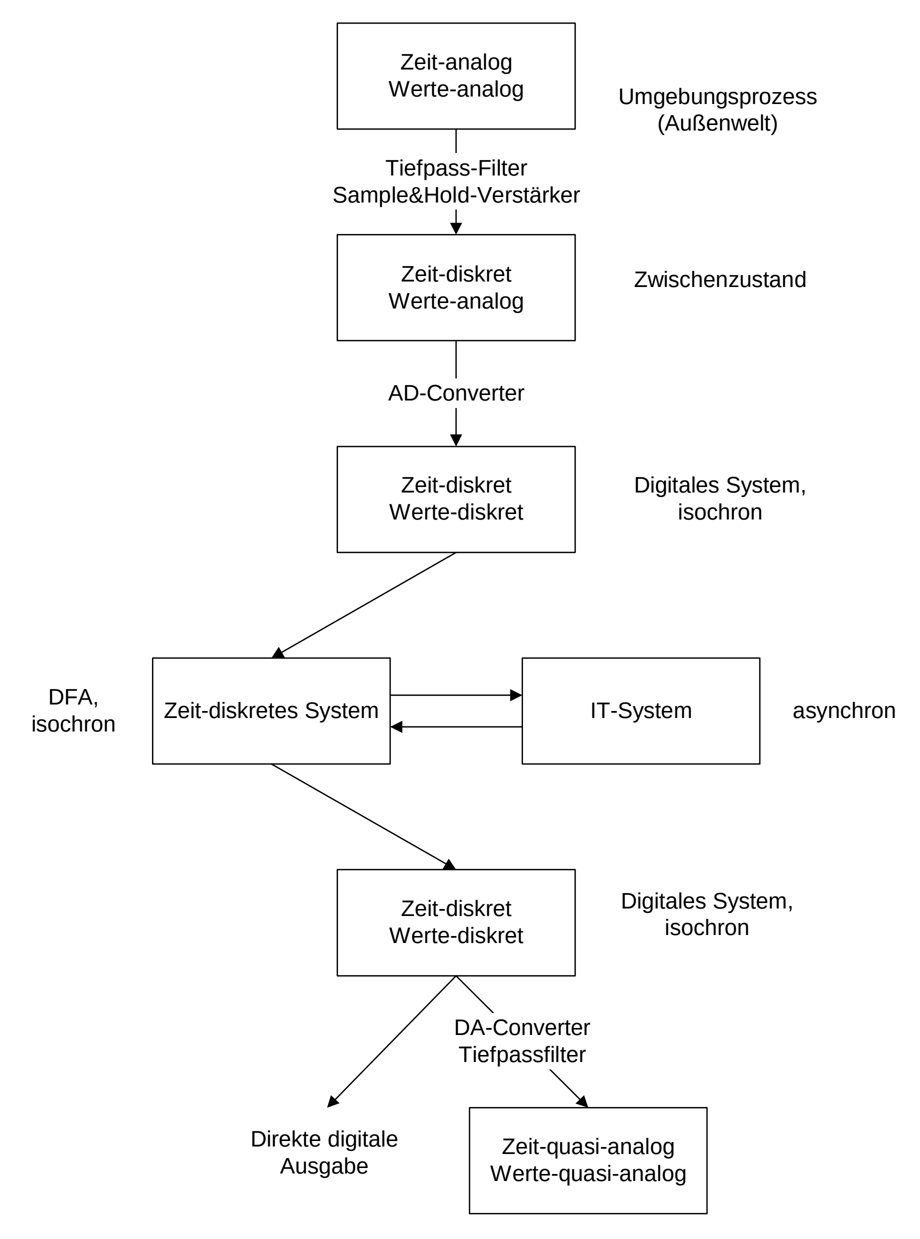


Bild 1.5 Übergänge zwischen den Zeitbindungen

In der Praxis sieht die Kopplung zwischen diesen drei Zeitbindungen so aus, dass Übergänge durch bestimmte Bausteine oder Vorgänge geschaffen werden. Bild 1.5 stellt dies zusammenfassend dar.

Hieraus lassen sich zwei Probleme identifizieren:

- Es gibt einen Informationsverlust beim Übergang zwischen der analogen und der Zeit- und Werte-diskreten Welt vor. Dieser Informationsverlust ist seit langem bekannt (Shannon, Abtasttheorem) und ausreichend behandelt.
- Im System liegt eine Kopplung zwischen isochronen und asynchronen Teilen vor. Die isochronen Teile behandeln den Umgebungsprozess mit gleicher Zeitbindung, während die asynchronen Systemteile ohne Bindung mit eigener Programmlaufzeit laufen, dennoch jedoch algorithmischen Bezug dazu haben. Diese Schnittstelle ist sorgfältig zu planen.

Die im letzten Aufzählungspunkt geforderte sorgfältige Planung der Schnittstelle führt dann zu den Echtzeitsystemen (→ 2), bei denen die Anforderungen an das IT-System so gestellt werden, dass das System auf einem gewissen Level wieder isochron arbeitet.

1.3.2 Weitere Randbedingungen für eingebettete Systeme

Die Zeit spielt in Embedded Systems aus dem Grund eine übergeordnete Rolle, weil der Rechner in eine Maschine eingebettet ist, deren Zeitbedingungen vorherbestimmt sind. Insofern hat die Zeit eine übergeordnete Bedeutung.

Aber: Es existieren noch weitere Randbedingungen, insbesondere für den Entwurfsprozess:

- Power Dissipation/Verlustleistung: Welche Durchschnitts- und/oder Spitzenleistung ist vertretbar, gefordert, nicht zu unterschreiten usw.?
- Ressourcenminimierung: Nicht nur die Verlustleistung, auch die Siliziumfläche, die sich in Kosten niederschlägt, soll minimiert werden.

Als vorläufiges Fazit kann nun gelten, dass die Entwicklung für eingebettete Systeme bedeutet, eine Entwicklung mit scharfen und unscharfen Randbedingungen durchzuführen.

2 Echtzeitsysteme

Dieses Kapitel dient dazu, die im vorangegangenen Kapitel bereits skizzierten Probleme der Integration der Zeit noch näher zu spezifizieren und vor allem die Lösungen aufzuzeigen. Dies führt zu den Echtzeitsystemen, und im ersten Teil dieses Kapitels werden Definitionen und Entwicklungsmethoden hierzu formuliert.

Die wirkliche Problematik beginnt genau dann, wenn mehrere Algorithmen nebenläufig zueinander zum Ablauf kommen. Dies ist Inhalt des zweiten Teils, in dem nebenläufige Systeme betrachtet werden.

2.1 Echtzeit

2.1.1 Definitionen um die Echtzeit

Die DIN 44300 des Deutschen Instituts für Normung beschreibt den Begriff Echtzeit wie folgt [Sch05]:

Definition 2.1:

Unter *Echtzeit* (*real time*) versteht man den Betrieb eines Rechensystems, bei dem Programme zur Verarbeitung anfallender Daten ständig betriebsbereit sind, derart, dass die Verarbeitungsergebnisse innerhalb einer vorgegebenen Zeitspanne verfügbar sind. Die Daten können je nach Anwendungsfall nach einer zeitlich zufälligen Verteilung oder zu vorherbestimmten Zeitpunkten anfallen.

Demgegenüber wird im *Oxford Dictionary of Computing* das Echtzeitsystem wie folgt beschrieben:

Definition 2.2:

Ein *Echtzeitsystem* (*real-time system*) ist ein System, bei dem der Zeitpunkt, zu dem Ausgaben vorliegen, bedeutend ist. Das liegt für gewöhnlich daran, dass die Eingabe mit einigen Änderungen der physikalischen Welt korrespondiert und die Ausgabe sich auf diese Änderungen beziehen muss. Die Verzögerung zwischen der Zeit der Eingabe und der Zeit der Ausgabe muss ausreichend klein für eine akzeptable "*Rechtzeitigkeit*" (*timeliness*) sein.

Echtzeitsysteme sind also Systeme, die korrekte Reaktionen innerhalb einer definierten Zeitspanne produzieren müssen. Falls die Reaktionen das Zeitlimit überschreiten, führt dies zu Leistungseinbußen, Fehlfunktionen und/oder sogar Gefährdungen für Menschen und Material.

Die Unterscheidung in harte und weiche Echtzeitsysteme wird ausschließlich über die Art der Folgen einer Verletzung der Zeitschranken getroffen:

Definition 2.3:

Ein Echtzeitsystem wird als *hartes Echtzeitsystem* (*hard real-time system*) bezeichnet, wenn das Überschreiten der Zeitlimits bei der Reaktion erhebliche Folgen haben kann. Zu diesen Folgen zählen die Gefährdung von Menschen, die Beschädigung von Maschinen, also Auswirkungen auf Gesundheit und Unversehrtheit der Umgebung.

Typische Beispiele hierfür sind einige Steuerungssysteme im Flugzeug oder im Auto, z.B. bei der Verbrennungsmaschine.

Definition 2.4:

Eine Verletzung der Ausführungszeiten in einem *weichen Echtzeitsystem* (*soft real-time system*) führt ausschließlich zu einer Verminderung der Qualität, nicht jedoch zu einer Beschädigung oder Gefährdung.

Beispiele hierfür sind Multimediasysteme, bei denen das gelegentlich Abweichen von einer Abspielrate von 25 Bildern/sek. zu einem Ruckeln o.ä. führt.

Als Anmerkung sei hier beigefügt, dass fast immer nur die oberen Zeitschranken aufgeführt werden. Dies hat seine Ursache darin, dass die Einhaltung einer oberen Zeitschranke im Zweifelsfall einen erheblichen Konstruktionsaufwand erfordert, während eine untere Schranke, d.h. eine Mindestzeit, vor der nicht reagiert werden darf, konstruktiv unbedenklich ist. Ein Beispiel für ein System, bei dem beide Werte wichtig sind, ist die Steuerung des Zündzeitpunkts bei der Verbrennungsmaschine: Dieser darf nur in einem eng begrenzten Zündintervall kommen.

2.1.2 Ereignissteuerung oder Zeitsteuerung?

Es stellt sich nun unmittelbar die Frage, wie die harten Echtzeitsysteme denn konzipiert sein können. Auf diese Frage wird im Kapitel 3.2 noch näher eingegangen, denn die Grundsatzentscheidung, welches Design zum Tragen kommen soll, hat natürlich erhebliche Konsequenzen für die gesamte Entwicklung.

Zwei verschiedene Konzeptionen, die in der Praxis natürlich auch gemischt vorkommen können, können unterschieden werden: *Ereignisgesteuerte* (*event triggered*) und *zeitgesteuerte* (*time triggered*) *Systeme*.

Ereignisgesteuerte Systeme werden durch Unterbrechungen gesteuert. Liegt an einem Sensor ein Ereignis (was das ist, muss natürlich definiert sein) vor, dann kann er eine *Unterbrechungsanforderung* (*interrupt request*) an den Prozessor senden und damit auf seinen Bedienungswunsch aufmerksam machen.

Definition 2.5:

Eine *asynchrone Unterbrechung* (*Asynchronous Interrupt Request*, *IRQ*) ist ein durch das Prozessor-externe Umfeld generiertes Signal, das einen Zustand anzeigt und/oder eine Behandlung durch den Prozessor anfordert. Dieses Signal ist nicht mit dem Programmlauf synchronisiert. Die Behandlung der Unterbrechung erfolgt

im Rahmen der *Interrupt Service Routine* (ISR), die für jede Unterbrechung im Softwaresystem definiert sein muss.

Bei zeitgesteuerten Systemen erfolgt keine Reaktion auf Eingabeereignisse, die Unterbrechungen werden lediglich durch einen, ggf. mehrere periodische Zeitgeber (Timer) ausgelöst. Sensoren werden dann vom Steuergerät aktiv abgefragt.

Dieses Verfahren hat den großen Vorteil, dass das Verhalten sämtlicher Systemaktivitäten zur Compilezeit vollständig planbar. Dies ist gerade für den Einsatz in Echtzeitsystemen ein erheblicher Vorteil, da á priori überprüft werden kann, ob Echtzeitanforderungen eingehalten werden. Dies wird in Abschnitt 3.2 genauer untersucht.

Der Vorgänger des zeitgesteuerten Designs wurde *Polling* genannt. Hierunter wird das ständige (quasi-zyklische), im Programm verankerte Abfragen von Prozesszuständen oder Werten verstanden, während das zeitgesteuerte Verfahren nicht ständig (also durch den Programmlauf bestimmte), sondern zu festgelegten Zeiten abfragt. Man beachte hierbei die Unterscheidung zwischen realer Zeit und Programmlaufzeit (→ 1.3.1).

Das Design dieser Zeitsteuerung muss allerdings sehr präzise durchgeführt werden, um die Ereignisse zeitlich korrekt aufzunehmen und zu verarbeiten. Ggf. müssen auch Zwischenpufferungen (z.B. bei einer schnellen Datenfolge) eingefügt werden. Um den zeitlichen Ablauf und seine Bedingungen quantifizieren zu können, seien folgende Zeiten definiert:

Definition 2.6:

Die *Latenzzeit* (*Latency Time*) ist diejenige vom Auftreten eines Ereignisses bis zum Start der Behandlungsroutine. Diese Zeit kann auf den Einzelfall bezogen werden, sie kann auch als allgemeine Angabe (Minimum, Maximum, Durchschnittswert mit Streuung) gewählt werden.

Definition 2.7:

Die *Ausführungszeit* (*Service Time*) ist die Zeit zur reinen Berechnung einer Reaktion auf ein externes Ereignis. In einem deterministischen System kann diese Zeit bei gegebener Rechengeschwindigkeit prinzipiell vorherbestimmt werden.

Definition 2.8:

Die *Reaktionszeit* (*Reaction Time*) ist diejenige Zeit, die vom Anlegen eines Satzes von Eingangsgrößen an ein System bis zum Erscheinen eines entsprechenden Satzes von Ausgangsgrößen benötigt wird.

Die Reaktionszeit setzt sich aus der Summe der Latenzzeit und der Ausführungszeit zusammen, falls die Service Routine nicht selbst noch unterbrochen wird.

Definition 2.9:

Die *Frist* (*Dead Line*) kennzeichnet den Zeitpunkt, zu dem die entsprechende Reaktion am Prozess spätestens zur Wirkung kommen muss. Diese Fristen stellen eine der wesentlichen Randbedingungen des Umgebungsprozesses dar.

Dies bedeutet also, dass zu jedem zu den Echtzeitkriterien zählendes Ereignis eine Frist definiert sein muss, innerhalb derer die Reaktion vorliegen muss. Folglich ist nicht die Schnelligkeit entscheidend, es ist Determinismus im Zeitsinn gefragt.

2.1.3 Bemerkungen zu weichen und harten Echtzeitsystemen

Die Konzeption eines harten Echtzeitsystems und vor allem der Nachweis dieser Fähigkeit ist außerordentlich schwierig, insbesondere, wenn man bedenkt, dass die Unterschiede im Laufzeitbedarf für einzelne Aufgaben sehr hoch sein können (für Fußball-spielende Roboter wird von 1:1000 berichtet). Es muss also auf den Maximalfall ausrichtet werden, wenn das System wirklich in jedem Fall in festgelegten Zeiten reagieren soll.

Man muss allerdings auch sagen, dass dieses Echtzeitkriterium aufweichbar ist (was auch z.B. von Anbietern der Echtzeit-Betriebssysteme gemacht wird):

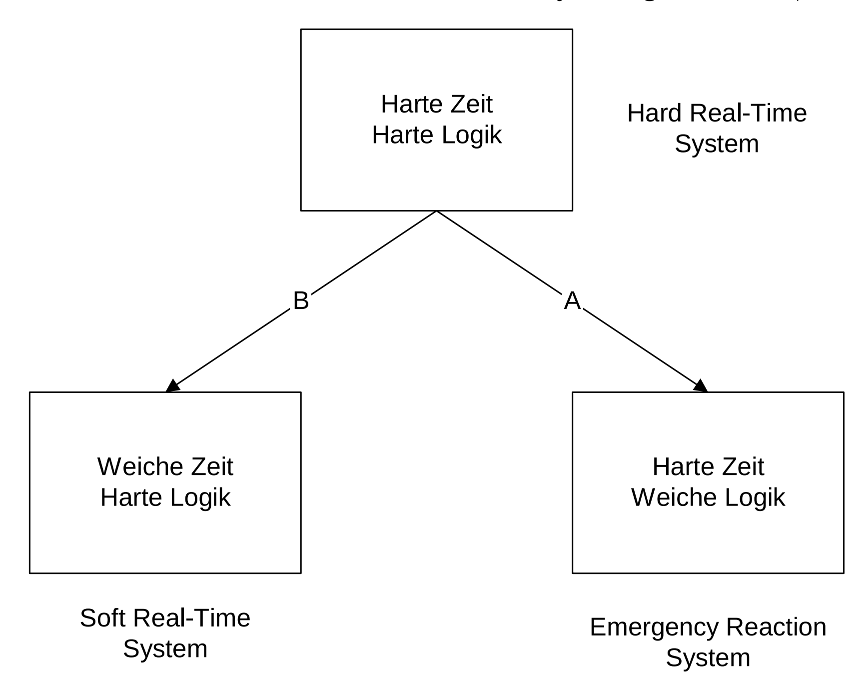


Bild 2.1 Darstellung verschiedener Applikationsklassen

Kann die vollständige, harte Reaktion nicht eingehalten werden, so bietet sich die Wege A und B in Bild 2.1 an. Weg A gilt dabei für Systeme bzw. Ereignisse, bei denen aus einer verspäteten Reaktion Schädigungen bis zur Zerstörung resultieren können. Hier wird nicht mit dem vollständig berechneten Ergebnis gehandelt, sondern mit einem ungefähren Wert, also eine Art rechtzeitige Notreaktion.

Weg B ist der gewöhnliche Ausweg. Hier werden Systeme vorausgesetzt, bei denen eine zeitliche Überschreitung zu einer Güteverminderung (Soft Degradation), nicht jedoch zu einer Schädigung führt. Wie bereits erwähnt bezeichnet man dies dann als *Soft Real-Time*, und dies wird gerne für Betriebssysteme genutzt.

2.2 Nebenläufigkeit

Nebenläufigkeit bildet das Grundmodell für Multiprocessing und Multithreading [Sch05]. Zwei Prozesse bzw. Threads sind dann nebenläufig, wenn sie unabhängig voneinander arbeiten können und es keine Rolle spielt, welcher der beiden Prozesse/ Threads zuerst ausgeführt oder beendet wird. Indirekt können diese Prozesse dennoch voneinander abhängig sein, da sie möglicherweise gemeinsame Ressourcen beanspruchen und untereinander Nachrichten austauschen.

Hieraus kann eine Synchronisation an bestimmten Knotenpunkten im Programm resultieren. Hier liegt eine Fehlerquelle, denn es kann hier zu schwerwiegenden Fehlern, *Verklemmungen* (*deadlocks*) und damit zu einem Programmabsturz kommen.

Die Hauptargumente, warum es trotz der Probleme (sprich: neue Fehlermöglichkeiten für Softwareentwickler) sinnvoll ist, Programme nebenläufig zu entwickeln, sind:

- Die Modellierung vieler Probleme wird dadurch vereinfacht, indem sie als mehr oder weniger unabhängige Aktivitäten verstanden werden und entsprechend durch Sprachkonstrukte umgesetzt werden können. Jede Aktivität kann dann isoliert betrachtet werden, nur die Kommunikation und Synchronisation ist zu beachten. Nebenläufigkeit führt hier zu einer abstrakteren Modellierung, und ob die entstandene Nebenläufigkeit dann wirklich zu einer gleichzeitigen Bearbeitung führt, ist nebensächlich.

  Ein Beispiel hierzu wird in Kapitel 4 behandelt, wo Messwertaufnahme und Auswertung in zwei miteinander gekoppelte, aber ansonsten getrennte Aktivitäten modelliert und auch implementiert werden.

- Die Anzahl der ausführenden Einheiten in einem Rechner kann durchaus > 1 sein. Im Zeitalter von Hardware/Software Co-Design, Multi- und Manyprozessorcores, konfigurierbaren Prozessoren, Prozessoren mit eigenem Peripherieprozessor usw. können Aufgaben auf verschiedene Teile abgebildet werden, und dazu müssen sie auch dergestalt modelliert sein. Hier wird die Performance

des Systems entscheidend verbessert, wenn die parallelen Möglichkeiten auch wirklich ausgenutzt werden.

2.2.1 Multiprocessing und Multithreading

Mit *Multitasking* wird allgemein die Fähigkeit von Software (beispielsweise Betriebssystemen) bezeichnet, mehrere Aufgaben scheinbar gleichzeitig zu erfüllen. Dabei werden die verschiedenen Tasks in so kurzen Abständen immer abwechselnd aktiviert, dass für den Beobachter der Eindruck der Gleichzeitigkeit entsteht. Man spricht hier auch oft von Quasi-Parallelität, aber mikroskopisch wird natürlich nichts wirklich parallel zueinander bearbeitet.

Doch was ist eine *Task*? Dies wird üblicherweise als allgemeiner Überbegriff für Prozesse und Threads (= Leichtgewichtsprozesse) genannt. Nun sind auch diese beiden schwer zu unterscheiden (zumindest präzise zu unterscheiden), aber meist reicht auch schon eine etwas unscharfe Definition.

Ein *Prozess* (*process*) ist ein komplettes, gerade ablaufendes Programm. Zu diesem Prozess gehören der gesamte Code und die statischen und dynamisch angelegten Datenbereiche einschließlich Stack, Heap und Register. Der Code wiederum kann mehrere Teile enthalten, die unabhängig voneinander arbeiten können. Solche Teile werden *Threads* (Aktivitätsfäden) genannt.

Ein Thread ist also ein Teil eines Prozesses, bestehend aus einem in sich geschlossenen Bearbeitungsstrang und einem recht minimalen eigenen Datenkontext. Letzterer wird benötigt, damit die Threads überhaupt parallel zueinander arbeiten können, und meist beschränkt sich dieser auf den Registersatz (des ausführenden Prozessors).

Welche Formen des Multiprocessing oder Multithreading gibt es denn? Das am häufigsten angewandte Konzept ist das *präemptive Multiprocessing*. Hier wird von einem Betriebssystem(kern) der aktive Prozess nach einer Weile verdrängt, zu Gunsten der anderen. Diese Umschaltung wird *Scheduling* genannt.

Die andere Form ist das *kooperative Multiprocessing*, das von jedem Prozess erwartet, dass dieser die Kontrolle an den Kern von sich aus zurückgibt. Letztere Version birgt die Gefahr in sich, dass bei nicht-kooperativen Prozessen bzw. Fehlern das gesamte System blockiert wird. Andererseits ist das kooperative Multiprocessing sehr einfach zu implementieren, auch innerhalb einer Applikation, daher wird dies als Beispiel in Kapitel 4 realisiert.

Beim Multithreading ist es ähnlich, wobei allerdings die Instanz, die über das Scheduling der Threads entscheidet, auch im Programm liegen kann (Beispiel: Java-Umgebung). Das Umschalten zwischen Threads eines Prozesses ist dabei wesentlich weniger aufwändig, verglichen mit Prozessumschaltung, weil im gleichen Adressraum verweilt wird. Allerdings sind auch die Daten des gesamten Prozesses durch alle Threads manipulierbar.

2.2.2 Prozesssynchronisation und –kommunikation

Die *Prozesssynchronisation* dient dem Ablauf der nebenläufigen Programmteile und ermöglicht eine Form der Wechselwirkung zwischen diesen. Das Warten eines Prozesses auf ein Ereignis, das ein anderer auslöst, ist die einfachste Form dieser Prozesssynchronisation (gleiches gilt auch für Threads).

Die Prozesskommunikation erweitert die Prozesssynchronisation und stellt somit dessen Verallgemeinerung dar. Hier muss es neben den Ereignissen auch Möglichkeiten geben, die Daten zu übertragen. Die praktische Implementierung ist dann z.B. durch ein Semaphoren/Mailbox-System gegeben: Über Semaphoren wird kommuniziert, ob eine Nachricht vorliegt, in der Mailbox selbst liegt dann die Nachricht. Für ein Multithreadingsystem kann dies direkt ohne Nutzung eines Betriebssystems implementiert werden, da alle Threads auf den gesamten Adressraum zugreifen können. Dies gilt nicht für Multiprocessingsysteme, hier muss ein Betriebssystem zur Implementierung der Mailbox und der Semaphoren verwendet werden.

Bei diesen Kommunikation wie auch der einfachen Synchronisation kann es zu Verklemmungen kommen. Eine Menge von Threads (Prozessen) heißt *verklemmt*, wenn jeder Thread (Prozess) dieser Menge auf ein Ereignis im Zustand ”blockiert“ wartet, das nur durch einen anderen Thread (Prozess) dieser Menge ausgelöst werden kann. Dies ist im einfachsten Fall mit zwei Threads (Prozessen) möglich: Jeder Thread wartet blockierend auf ein Ereignis des anderen.

Im Fall der Prozess- oder Threadkommunikation kann dies gelöst werden, indem nicht-blockierend kommuniziert wird: Die Threads (Prozesse) senden einander Meldungen und Daten zu, warten aber nicht darauf, dass der andere sie auch abholt. Am Beispiel in Kapitel 4 wird gezeigt, dass dies auch notwendig für die Echtzeitfähigkeit ist, allerdings sollte nicht übersehen werden, dass hierdurch Daten auch verloren gehen können.

2.2.3 Grundlegende Modelle für die Nebenläufigkeit

Bezüglich der Zeit für das Aufbauen der Kommunikation zwischen zwei Prozessen (Threads) gibt es drei Grundannahmen: *Asynchron*, *perfekt synchron* (mit Null-Zeit) und *synchron* (mit konstanter Zeit). Asynchrone Kommunikation bedeutet in diesem Fall, dass die Kommunikationspartner sozusagen zufällig in Kontakt treten (wie Moleküle in einem Gas) und dann wechselwirken. Dieses Modell, als *chemisches Modell* bezeichnet, ist daher nichtdeterministisch und für eingebettete Systeme unbrauchbar.

> Anmerkung: Spricht man im Zusammenhang von Network-on-Chip (NoC) von asynchroner Kommunikation, so ist damit selbst-synchronisierende Kommunikation gemeint. Für RS232, auch eine ”asynchrone“ Schnittstelle, bedeutet asynchron, dass der Beginn einer Aussendung für den Empfänger spontan erfolgt. Auf höherer Ebene ist diese Kommunikation natürlich nicht zufällig, sondern geplant.

Das *perfekt synchrone Modell* geht davon aus, dass Kommunikation keine Zeit kostet, sondern ständig erfolgt. Dies lehnt sich an die Planetenbewegung an, wo die Gravitation untereinander und mit der Sonne zu den Bahnen führt, und wird deshalb auch *Newtonsches Modell* genannt. Die so genannten *synchronen Sprachen* basieren auf diesem Modell.

Das dritte Modell, das *synchron*, aber mit konstanter Zeitverzögerung kommuniziert, wird auch *Vibrationsmodell* genannt. Dieser Name entstammt der Analogie zur Kristallgitterschwingung, bei der eine Anregung sich über den Austausch von Phononen fortpflanzt.

Wozu dienen diese Kommunikationsmodelle? Der Hintergrund hierzu besteht darin, Kommunikation und Betrieb in nebenläufigen, ggf. auch verteilten Systemen modellieren zu können. Die Annahme einer perfekt synchronen Kommunikation beinhaltet eigentlich nicht, dass ”Null-Zeit“ benötigt wird, vielmehr ist die Zeit zur Bestimmung eines neuen Zustands im Empfänger kleiner als die Zeitspanne bis zum Eintreffen der nächsten Nachricht. Dies bedeutet, dass sich das gesamte System auf diese Meldungen synchronisieren kann.

3 Design von eingebetteten Systemen

Dieses Kapitel dient dem Zweck, den Zusammenhang zwischen den Systemen, die programmiert werden können, den Entwurfssprachen und der in Kapitel 1 bereits diskutierten Randbedingung *Echtzeitfähigkeit* darzustellen.

Diese Diskussion soll konstruktiv gestaltet werden, d.h. weniger theoretische Konzepte stehen im Vordergrund, vielmehr sollen Lösungsmöglichkeiten und Design Pattern (Architekturmuster) aufgezeigt werden. Zu diesem Zweck werden Ansätze zur Lösung des Echtzeitproblems diskutiert, und zwar in zwei Abschnitten: Abschnitt 3.1 diskutiert, wie die reale Zeit mit dem Ablauf im Mikroprozessor gekoppelt werden kann, Abschnitt 3.2 ist dann dem Systemdesign gewidmet. Im Anschluss daran folgt eine Einführung in Softwareentwicklungssprachen.

3.1 Ansätze zur Erfüllung der zeitlichen Randbedingungen

Gerade in eingebetteten Systemen ist der entscheidende Zeitbegriff derjenige der 'Reaktionszeit', der mit dem deterministischen Echtzeitverhalten des Systems korreliert. Hier geht es nicht um Einsparungspotenzial, sondern um die Erfüllung der zeitlichen Randbedingungen. Um dies zu erreichen, bieten sich 'Design Pattern' an, die in den folgenden Abschnitten dargestellt werden sollen.

3.1.1 Technische Voraussetzungen

Zunächst müssen einige Voraussetzungen für die hier dargestellten Ansätze erläutert werden. Als technische Basis sei ein *Mikroprozessor-basiertes Rechnersystem* angenommen, das kein Betriebssystem und somit keinen Scheduler (= Einheit zur Rechenzeitvergabe an unterschiedliche Tasks) zur Verfügung hat. Diese Einschränkung kann prinzipiell jederzeit aufgehoben werden, allerdings lassen sich an dem Betriebssystem-losen System die Einzelheiten zum Systemdesign präziser darstellen.

In der Praxis werden solche Systeme gerne als „kleine" Systeme eingesetzt. Es ist das Ziel dieses Unterkapitels und des Beispiels aus Kapitel 4, ein Softwaredesign vorzustellen, dass dennoch aus mehreren Teilen besteht und ein Applikations-internes Scheduling enthält.

Weiterhin soll der Begriff *Thread* hier in erweitertem Sinn genutzt werden. Wie in Abschnitt 2.2.1 dargestellt stellt ein Thread einen so genannten Programmfaden dar, der innerhalb eines (Software-)Prozesses abläuft bzw. definiert ist. Diese Definition ist vergleichsweise schwammig.

Für die Zwecke dieses Skripts sei der Begriff *Thread* erweitert. Ohne Betriebssystem existiert nur ein Prozess, der das einzige Programm, das abläuft, darstellt. Innerhalb dieses Prozesses ist ein Thread wie folgt definiert:

Definition 3.1:

Ein *Thread* ist ein in sich geschlossener Programmteil, der mit anderen, nicht zu diesem Programmteil gehörenden Teilen des Prozesses nur indirekt, d.h. nicht über Funktions- oder Methodenaufrufe kommuniziert.

Diese Definition wird sich im Laufe der nächsten Abschnitte bis hin zur Darstellung eines Applikations-internen Schedulers als sehr praktikabel erweisen.

3.1.2 Zeit-gesteuerte Systeme (Time-triggered Systems)

Eine Möglichkeit, den realen Bezug zwischen Realzeit und Programmlaufzeit zu schaffen, besteht darin, eine *feste Zeitplanung* einzuführen. Hierzu müssen natürlich alle Aufgaben bekannt sein.

Weiterhin müssen folgende Voraussetzungen gelten:

- Die Verhaltensweisen des Embedded Systems und des Prozesses müssen zur *Übersetzungszeit* (*compile time*) vollständig definierbar sein.
- Es muss möglich sein, eine gemeinsame Zeit über alle Teile des Systems zu besitzen. Dies stellt für ein konzentriertes System kein Problem dar, bei verteilten, miteinander vernetzten Systemen muss aber diesem Detail erhöhtes Augenmerk gewährt werden.
- Für die einzelnen Teile des Systems, also für jeden Thread, müssen exakte Werte für das Verhalten bekannt sein. Exakt heißt in diesem Zusammenhang, dass die Zeiten im Betrieb nicht überschritten werden dürfen. Es handelt sich also um eine Worst-Case-Analyse, die mit Hilfe von Profiling, Simulation oder einer exakten Laufzeitanalyse erhalten werden.

Hieraus ergibt sich dann ein planbares Verhalten. Man baut dazu ein *statisches* Scheduling (= Verteilung der Rechenzeit zur Compilezeit) auf, indem die Zykluszeit (= Gesamtzeit, in der aller Systemteile einmal angesprochen werden) aus dem Prozess abgeleitet wird.

Die praktische Ausführung eines Zeit-gesteuerten Systems kann dabei auf zwei Arten erfolgen: Auslösung durch Timer-Interrupt und ein kooperativer Systemaufbau:

- Beim Aufbau mit Hilfe von Timer-Interrupts wird ein zyklischer Interrupt (→ Definition 2.5) aufgerufen. Dies ist zwar auch eine Art Ereignis-Steuerung dar, sie ist aber geplant und streng zyklisch auftretend. In der Interrupt-Service-Routine (→ Definition 2.5) werden dann aller Prozesszustände abgefragt und entsprechende Reaktionsroutinen aufgerufen.

- Beim kooperativen Systemaufbau ist jeder Thread verpflichtet, eine Selbst-Unterbrechung nach einer definierten Anzahl von Befehlen einzufügen. Diese Unterbrechung ist als Aufruf eines Schedulers implementiert, dieser ruft dann einen weiteren Thread auf. Dieses Verfahren ist unschärfer und aufwendiger (die Zeiten müssen festgelegt werden), sodass meist die erste Variante bevorzugt wird.

Innerhalb der entstandenen Zykluszeit kann dann das Gefüge der Aufgaben verteilt werden. Im einfachsten Fall eines Zyklus, d.h. einer kritischen Aufgabe, müssen folgende Ungleichungen gelten:

$$t_{cycle} \leq t_{critical} \quad (3.1)$$

$$t_{thread} \leq t_{cycle} \quad (3.2)$$

Mit $t_{critical}$ ist hierbei die systemkritische Zeit angenommen, die für ein ordnungsgemäßes Arbeiten nicht überschritten werden darf. Diese Zeit wird durch den Prozess definiert und entspricht etwa der maximal möglichen Reaktionszeit. Zu einer genaueren Herleitung siehe 3.2.1.

Ungleichung (3.2) kann auch mehrere Threads enthalten, die ggf. sogar mehrfach berücksichtigt werden, weil sie beispielsweise mehrfach in einem Zyklus vorkommen müssen. Für diese Threads kann eine andere systemkritische Zeit gelten, und diesem Umstand kann man durch den (zeitlich verteilten) Mehrfachaufruf Rechnung tragen.

Die Zeitdefinition im Mikrocontroller kann durch einen Timer erfolgen, dieser Timer stellt eine Hardwareeinheit dar, die Takte zählen kann. Durch die Kopplung des Takts mit der Realzeit aufgrund der fest definierten Schwingungsdauer ergibt sich hierdurch ein Zeitgeber bzw. –messer.

Die exakte Bestimmung der Zykluszeit t_{cycle} wird in Abschnitt 3.2.1 beschrieben. Hier sind ggf. mehrere Bedingungen zu berücksichtigen. Die Berechnungszeit t_{thread} aus Ungleichung (3.2) kann durch die Bestimmung der Worst-Case-Execution-Time (WCET) in Kombination mit der Worst-Case-Interrupt-Disable-Time (WCIDT) berechnet bzw. geschätzt werden (→ 3.2.3).

Diese Variante hat folgende Vor- und Nachteile:

+ Garantierte Einhaltung kritischer Zeiten
+ Bei verteilten Systemen Erkennung von ausgefallenen Teilen (durch Planung von Kommunikation und Vergleich in den anderen Systemteilen)
– Das System muss hoch dimensioniert werden, weil für alle Teile die Worst-case-Laufzeiten angenommen werden müssen.
– Die Einbindung zeit<u>un</u>kritischer Teile erfolgt entweder unnötig im Scheduling, oder das System wird durch die Zweiteilung komplexer.

– Die Kombination mehrerer, Zeit-gesteuerter Tasks kann sich als sehr aufwendig erweisen, falls die einzelnen Zeitabschnitte in ungünstigem Verhältnis zueinander liegen (siehe nächsten Abschnitt).

3.1.3 Kombination mehrerer Timer-Interrupts

Als nächstes muss die Kombination mehrerer Aufgaben mit Zeitbindung diskutiert werden. Grundsätzlich ist es natürlich möglich, mehrere (unterschiedlich laufende) Zeitsteuerungen durch mehrere Timer-Interrupts durchzuführen. Beispiele hierfür sind die Kombination mehrerer Schnittstellen, etwa RS232 und I²C-Bus, die mit unterschiedlichen Frequenzen arbeiten, sowie die Kombination aus Messwertaufnahme und serieller Schnittstelle.

In diesem Fall wird für jeden Timer die entsprechende Zeitkonstante gewählt, also etwa die Zeit, die zwischen zwei Messungen oder zwei Transmissionen liegt (→ 3.2.1). Das Problem, das sich hierbei stellt, ist die zufällige zeitliche Koinzidenz mehrerer Interrupts, die behandelt werden muss. Das gleichzeitige oder doch sehr kurz aufeinander folgende Eintreffen der Requests bedeutet, dass die Behandlung eines Vorgangs gegenüber dem zweiten zurückgestellt wird. Dies muss zwangsläufig in jeder Kombination möglich sein, da nichts vorbestimmbar ist.

Ein anderer Weg ist ggf. einfacher zu implementieren: Alle Teilaufgaben, die zyklisch auftreten, werden in einer einzigen ISR, die von einem zyklisch arbeitenden Timer aufgerufen wird, zusammengefasst. Die Probleme, die dabei auftreten, liegen weniger im grundsätzlichen Design als vielmehr darin, mit welcher Frequenz bzw. mit welchem Zeitwert die ISR aufgerufen wird.

Während bei einer einzigen Aufgabe mit streng zyklischem Verhalten die Wahl einfach ist – die Zeitkonstante, die zwischen zwei Messungen oder zwei Transmissionen liegt, wird als der Timerwert gewählt –, muss nunmehr der größte gemeinsamen Teiler (ggT) der Periodenzeiten als Zeitwert gewählt werden.

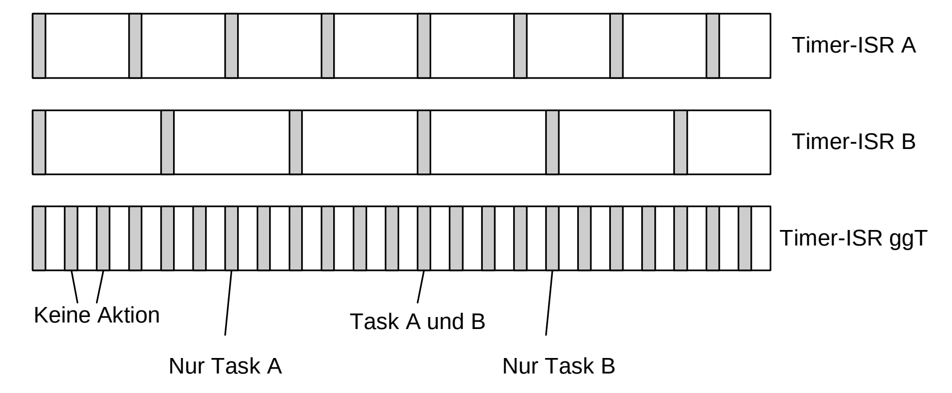


Bild 3.1 Zusammenfügen mehrerer Zeit-gesteuerter ISR zu einer Routine

Die ggT-Methode (Bild 3.1) ist so vorteilhaft, weil zu Beginn einer Timer-ISR bestimmt werden kann, was alles (und auch in welcher Reihenfolge) behandelt werden soll. Hierdurch lassen sich auch Zeitverschiebungen planen bzw. bestimmen. Andererseits kann der ggT-Ansatz sehr schnell in ein nicht-lauffähiges System münden. Die Anzahl der ISR pro Zeiteinheit kann stark zunehmen (→ Bild 3.1), und jeder Aufruf einer ISR erfordert einen zeitlichen Overhead, auch wenn keine weitere Routine darin abläuft. Als Faustregel sollte man mit mindestens 10 – 20 Befehlsausführungszeiten rechnen, die für Interrupt-Latenzzeit, Retten und Restaurieren von Registern und den Rücksprung in das Programm benötigt werden. In einem System, das 1 µs Befehlsausführungszeit hat und alle 200 µs unterbrochen wird, sind das aber bereits 5 – 10 % der gesamten Rechenzeit, die unproduktiv vergehen. Daher sollte, soweit dies möglich ist, die Periode so gewählt sein, dass der ISR-Overhead klein bleibt (< 5%).

Im Idealfall besteht darin, die Zykluszeiten gegenseitig anzupassen, so dass der ggT gleich dem kürzesten Timerwert ist. Dies führt zumindest zu einem System, das keine ISR-Aufrufe ohne Netto-Aktion (wie in Bild 3.1 dargestellt) hat.

3.1.4 Flexible Lösung durch integrierte Logik

Die Tatsache, dass durch die Wahl des ggT aller Zykluszeiten als die einzige Zykluszeit im Allgemeinen ”leere“ Unterbrechungen erzeugt werden, lässt sich dadurch umgehen, dass man von der periodischen Erzeugung abgeht und nun eine bedarfsgerechte Generierung einführt.

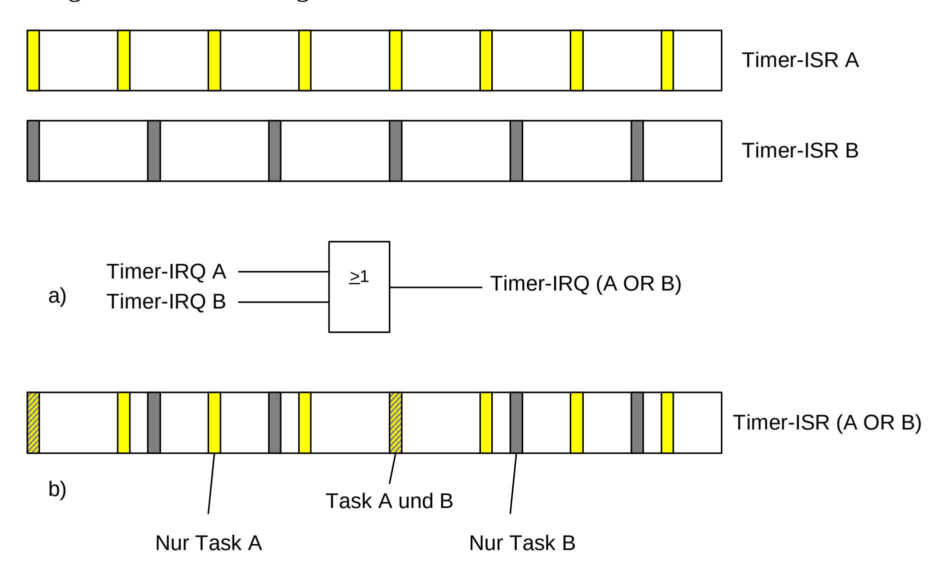


Bild 3.2 Zusammenfassung zweier Unterbrechungsquellen mittels Hardware
a) Verknüpfung der IRQ-Signale b) resultierendes Timingschema

Dies ist durch die Belassung bei mehreren Timern und anschließende OR-Verknüpfung der Unterbrechungssignale zu erreichen, wie Bild 3.2 darstellt. Damit ist dann ein effizientes Timingschema für die Unterbrechungen erzeugt, und die Unterbrechungsroutine würde unterscheiden, welche Aktionen durchzuführen wären.

Dies kann beliebig ausgestaltet werden, und sehr komplexe Interrupt-Schemata können erzeugt werden. Allerdings bleibt festzustellen, dass übliche Mikrocontroller die hierzu notwendige Hardware nicht enthalten, nur Derivate mit umfangreicher Peripherie bieten meist Timer-Arrays mit (begrenzter) Kombinationsfähigkeit an. Diese Form der Lösung bleibt damit meist den rekonfigurierbaren Prozessoren (Mikroprozessor + programmierbare Logik) bzw. der Zusammenstellung solcher Komponenten auf Boardlevel vorbehalten.

3.1.5 Ereignis-gesteuerte Systeme (Event-triggered Systems)

Timersignale stellen zwar auch eine Unterbrechung des üblichen Programmlaufes dar, allerdings ist dies grundsätzlich planbar, während Unterbrechungen aus dem Prozessumfeld nicht planbar sind.

In einem Ereignis-gesteuerten System reagiert das Gesamtprogramm auf die Ereignisse des Prozesses. Insbesondere werden die Prozesszustände nicht zyklisch abgefragt, sondern es werden Zustandsänderungen an den Prozessor per IRQ gemeldet.

Diese Form der Systemauslegung, die selten in reiner Form auftritt, bedingt natürlich einen vollkommen anderen Systemansatz:

1. Der Prozess muss mit exklusiver Hardware ausgestattet sein, die ein Interface zum Prozessor bildet. Diese Hardware muss Zustandsänderungen erkennen und per IRQ zum Prozessor signalisieren.
2. Im Prozessor und (höchstwahrscheinlich) dem Interrupt-Request-Controller muss ein Priorisierungssystem festgelegt werden, das die IRQs in ein Prioritätssystem zwingt und entsprechend behandelt. Zu dieser Priorisierungsstrategie gehören auch Fragen wie ”Unterbrechungen von Unterbrechungs-Serviceroutinen“.
3. Es ist wahrscheinlich, dass neben den IRQ-Serviceroutinen (ISR) auch weitere, normale Programme existieren. Dies erfordert eine Kopplung zwischen ISR und Hauptprogramm.

Hieran ist zu erkennen, dass die Planung dieses Systems alles andere als einfach ist. Insbesondere stecken Annahmen in dem IRQ-Verhalten des Prozesses, die Aussagen zur Machbarkeit erst ermöglichen, so z.B. eine maximale Unterbrechungsrate.

Unter bestimmten Umständen kann die Erfüllung der Realtime-Bedingungen äquivalent zum Zeit-gesteuerten Design garantiert werden. Die Bedingungen hierzu sind:

- keine ISR kann unterbrochen werden (dies ist im Zeit-gesteuerten Design implizit eingeschlossen)
- für jeden IRQ ist eine maximale Frequenz des Auftretens und eine maximale Reaktionszeit gegeben

dann gilt verhält sich das Ereignis-gesteuerte System im Maximalfall entsprechend wie das Zeit-gesteuerte System mit gleichen Zykluszeiten und kann entsprechend ausgelegt werden. Scheinbar spart man Systemkapazität, weil die maximale Auftrittsfrequenz nicht unbedingt eintreten muss, für das Design und die Systemauslegung muss jedoch mit dem Worst-Case gerechnet werden. Insgesamt gilt hier also:

+ Bei 'weicher' Echtzeit ist eine gute Anpassung an die real benötigten Ressourcen möglich.
+ Die Einbindung zeitunkritischer Teile ist sehr gut möglich, indem diese im Hauptprogramm untergebracht werden und so automatisch die übrig bleibende Zeit zugeteilt bekommen.
– Die Bestimmung und der Nachweis der Echtzeitfähigkeit sind außerordentlich schwierig.
– Bei harten Echtzeitbedingungen droht eine erhebliche Überdimensionierung des Systems.
– Die Annahme der maximalen IRQ-Frequenz ist meist eine reine Annahme, die weder überprüfbar und automatisch einhaltbar ist. So können z.B. prellende Schalterfunktionen IRQs mehrfach aufrufen, ohne dass dies in diesem System vermieden werden kann.

Gerade der letzte Punkt ist kritisch, denn die Annahme kann durch die Praxis falsifiziert werden. Hierfür hilft eine Variante im nächsten Abschnitt, aber es bleibt immer noch die Aussage, dass die Systemauslegung zur Erreichung der Echtzeitfähigkeit nicht oder nur marginal wenig gemindert werden kann.

3.1.6 Modified Event-driven Systems

Einer der wesentlichen Nachteile der Ereignis-gesteuerten Systeme liegt in der Annahme, dass die asynchronen Ereignisse mit einer maximalen Wiederholungsfrequenz auftreten. Diese Annahme ist notwendig, um die Machbarkeit bzw. die reale Echtzeitfähigkeit nachweisen zu können.

Andererseits zeigen gerade die Ereignissteuerungen eine bessere Ausnutzung der Rechenleistung, weil sie den Overhead der Zeitsteuerung nicht berücksichtigen müssen. Es stellt sich die Frage, ob ein Ereignis-gesteuertes System nicht so modifiziert werden kann, dass die Vorteile bleiben, während die Nachteile aufgehoben oder gemildert werden.

Der Schlüssel hierzu liegt in einer Variation der Hardware zur Übermittlung und Verwaltung der Interrupt Requests. Mit Hilfe eines spezifisch konfigurierten Ti-

mers pro Interrupt-Request-Kanal im IRQ-Controller kann jeglicher Interrupt nach Auftreten für eine bestimmte Zeit unterdrückt werden. Bild 3.3 zeigt das Blockschaltbild des hypothetischen IRQ-Controllers.

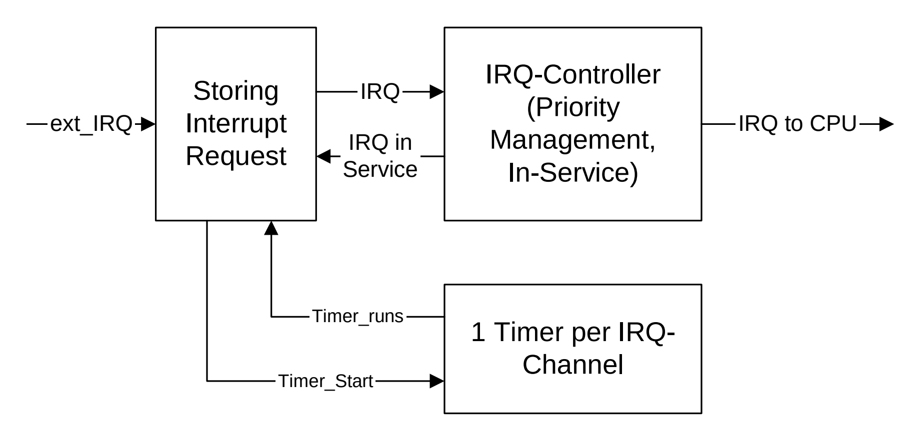


Bild 3.3 Modifizierter IRQ-Controller

Die vorgesehene Wirkungsweise des Timers ist diejenige, dass weitere IRQ-Signale, die vor dem Start der ISR auftreten, weder berücksichtigt noch gespeichert werden, während Signale, die nach dem Start der ISR, aber vor dem Ablauf des Timers eintreffen, gespeichert werden, jedoch vorerst keine Aktion hervorrufen. Diese etwas aufwendige Definition dient dazu, ein Maximum an Systemintegration zu erreichen.

Die Unterdrückung aller weiteren IRQ-Signale bis zum Eintritt in die ISR entspricht dabei der gängigen Praxis, mehrfache IRQs nur einmalig zu zählen. Die aktionslose Speicherung nach dem Eintritt lässt dabei keinen IRQ verloren gehen, und nach dem Timerablauf wird der gespeicherte IRQ aktiv (und startet den Timer sofort neu).

Diese Funktionsweise zwingt die asynchronen Interrupt Requests in ein Zeitschema, für das das Rechnersystem ausgelegt wird. Sind alle IRQs mit diesem Verfahren der Beschränkung der Wiederholungsfrequenz ausgestattet, können für alle Teile des Systems die maximalen Bearbeitungszeiten berechnet werden. Das modifizierte Ereignis-gesteuerte System wird hierdurch genauso deterministisch wie das Zeit-gesteuerte System mit dem Zusatz, dass keinerlei Pollingaktivitäten ablaufen müssen und ungenutzte Ereignisrechenzeiten den zeitunkritischen Programmteilen zugute kommen.

Für das Modified Event-Triggered System sind folgende Vor- und Nachteile anzugeben:

\+ Deterministische Berechenbarkeit des Zeitverhaltens, wie beim Time.-triggered System.

+ Ungenutzte Zeit, die für Ereignisse vorgesehen war, wird an zeitunkritische Teile des Systems weitergegeben, es entsteht kein Overhead.
+ Verfahren ist mit Einschränkung auch auf Netzwerke übertragbar, indem die einzelnen Knoten maximale Senderaten bekommen und eine unabhängige Hardware dies überwacht. Die Einschränkungen betreffen den Netzzugang, hier sind nur Collision-Avoidance-Verfahren (z.B. CAN) zulässig.
– Die Systemauslegung orientiert sich weiterhin an Worst-Case-Schätzungen.
– Alle IRQs zählen zu der Reihe der deterministischen Ereignisse, die auf diese Weise behandelt werden müssen; Ereignisse mit beliebigen Reaktionszeiten oder 'weichen' Behandlungsgrenzen existieren nicht.
– Die variierte Hardware ist derzeit nicht erhältlich, muss also selbst definiert werden (z.B. in programmierbarer Hardware).

3.1.7 Modified Event-triggered Systems with Exception Handling

Während die Einschränkung der tatsächlichen IRQ-Raten den Determinismus in Event-triggered Systemen erzeugen kann, ist das Problem der maximalen Systemauslegung hierdurch noch nicht gelöst oder wesentlich gemildert. Die Einschränkung aus 3.1.6 schafft nur den Determinismus, der zuvor lediglich angenommen werden konnte.

Die Überdimensionierung eines Systems rührt von der erfahrungsgemäß großen Diskrepanz zwischen Worst-Case-Schätzung und realistischen Normalwerten. Natürlich lässt sich ein System nicht auf Erfahrungswerten so aufbauen, dass es zugleich auch beweisbar deterministisch ist.

Folgender Weg bietet unter bestimmten Umständen eine Möglichkeit, einen guten Kompromiss zwischen beweisbarer Echtzeitfähigkeit und Dimensionierung des Systems zu finden. Dieser Ansatz wird als '*Modified Event-triggered System with Exception Handling*' bezeichnet.

Folgende Voraussetzungen sind notwendig, um einen Interrupt Request, der zu der deterministischen Ereignisreihe gehört, in eine zweite Kategorie, die mit *Ereignisreihe mit variierter Reaktionsmöglichkeiten* bezeichnet wird, zu transferieren:

- Grundsätzlich wird das System als Ereignis-gesteuert so ausgelegt wie in den vorangegangenen zwei Abschnitten beschrieben.
- Für das ausgewählte Ereignis muss eine Notreaktionsmöglichkeit existieren, beispielsweise ein allgemein gültiger, ungefährer Reaktionswert, der in einer gesonderten Reaktionsroutine eingesetzt werden kann oder
- Die Berechnungszeit für das ausgewählte Ereignis kann erweitert werden.

Mit Hilfe einer nochmalig erweiterten Hardwareunterstützung im Prozessor und im Interrupt Request Controller kann dann ein erweitertes IRQ-Handling eingeführt werden. Die ergänzende Hardware ist in Bild 3.4 dargestellt.

Die Ergänzung besteht darin, einen weiteren Timer pro Interrupt Request im IRQ-Controller vorzusehen. Dieser Timer wird mit jeder IRQ-Speicherung gestartet und enthält einen Ablaufwert, der der maximalen Reaktionszeit entspricht. Ist die Interrupt-Service-Routine beendet, so muss der Timer natürlich gestoppt werden, z.B. explizit durch zusätzliche Befehle oder implizit durch Hardwareerweiterung in der CPU (erweiterter RETI-Befehl, Return from Interrupt mit IRQ-Nummer).

Der Ablauf eines solchen Timers soll dann eine Time Exception (= Interrupt Request mit hoher Priorität) auslösen und damit eine Ausnahmebehandlung initiieren. Es ist hierbei möglich, alle derart ergänzten IRQs mit einer Time Exception zu versehen und damit in einer Routine zu behandeln.

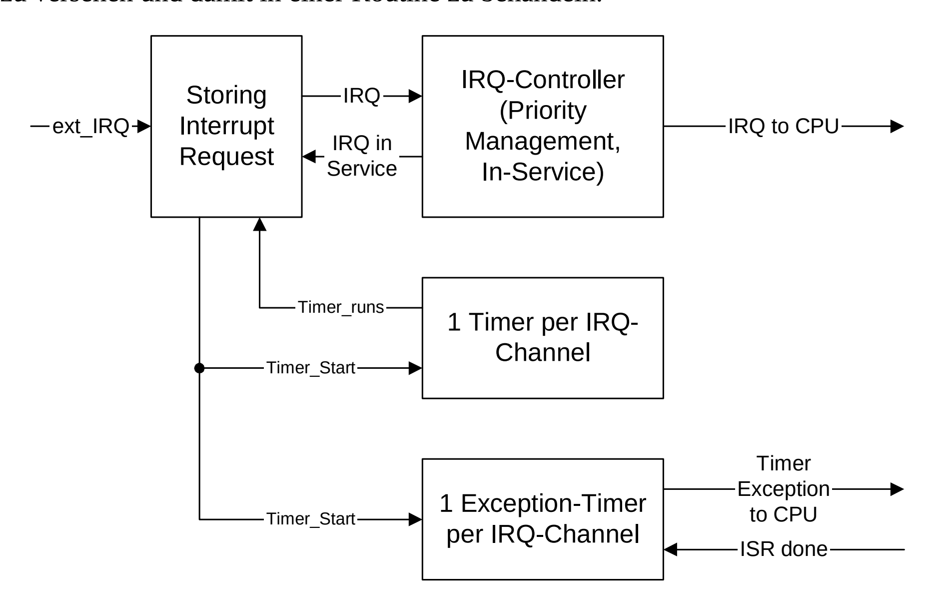


Bild 3.4 Erweiterter IRQ-Controller mit Time Exception

Die Ausnahmeroutine kann dann von fallweise entscheiden, wie vorzugehen ist. Existiert ein Notwert, der z.B. eine bereits berechnete, ungefähre Näherung (aber nicht den exakten Wert) darstellt, kann dieser eingesetzt und die Service-Routine damit für diesen Fall beendet werden. Es kann auch entschieden werden, einen weiteren Zeitabschnitt zu durchlaufen, falls dies für dieses Ereignis möglich ist.

Gerade die Möglichkeit, Näherungswerte einzusetzen, stellt ein mächtiges Instrumentarium dar, um harte Echtzeit bei 'weicher' Logik zu erhalten. Dies ist bei bisherigen Verfahren nur mit sehr großem Rechenaufwand möglich, Aufwand, der gerade aus Zeitknappheit entfallen muss.

Für diesen Ansatz zur Erreichung eines echtzeitfähigen Systems können folgende Vor- und Nachteile angegeben werden:

+ Deterministische Berechenbarkeit des Zeitverhaltens, wie beim Time.-triggered und Modified Event-triggered System.
+ Ungenutzte Zeit, die für Ereignisse vorgesehen war, wird an zeitunkritische Teile des Systems weitergegeben, es entsteht kein Overhead.
+ Die Systemauslegung orientiert sich nicht mehr an Worst-Case-Schätzungen mit vollständigem Rechenweg, sondern für eine deterministische Auslegung nur noch bis zu den Näherungswerten.
– Komplexe Klassifizierung der Ereignisse notwendig: Welche Events sind immer vollständig durchzurechnen, welche können Näherungen haben, für welche sind Zeiterweiterungen (in Grenzen) zulässig?
– Softwareunterstützung ist derzeit nicht erhältlich, folglich ist alles Handdesign.
– Die erweiterte Hardware ist derzeit nicht erhältlich.

3.2 Bestimmung der charakteristischen Zeiten im System

Nachdem nunmehr die Grundzüge eines zeitbasierten Software-Engineerings aufgezeigt sind müssen die charakteristischen Zeiten im (späteren) System definiert bzw. bestimmt werden. Zu diesen Zeiten zählen die Zykluszeit, die aus dem Außenprozess heraus definiert sein muss, dann die Worst-Case-Execution-Time (WCET) und die Worst-Case-Interrupt-Disable-Time (WCIDT), die berechnet bzw. geschätzt werden müssen, nachdem die Routinen geschrieben sind, sowie der Nachweis der Echtzeitfähigkeit für das ganze System.

3.2.1 Zykluszeiten

In diesem Abschnitt wird von der Annahme ausgegangen, dass pro betrachtetem Thread exakt eine Zykluszeit auftritt. Dies bedeutet im Einzelnen:

- Threads ohne Zeitbindung, die dementsprechend zeitlich unkritisch sind, werden nicht betrachtet
- Ein Thread kann zwar mehrere Aufgaben beinhalten, die dann aber der gleichen Zykluszeit unterworfen sind.

Für die Bestimmung der Zykluszeit eines Threads sind ferner folgende Zeiten des Außenprozesses wichtig:

Definition 3.2:

Die *maximal erlaubte Reaktionszeit* ***Tmx*** ist die maximal mögliche Wartezeit vom Eintreten eines Ereignisses bis zur vollständigen Reaktion darauf (siehe auch Definition 2.8). Das Eintreten des Ereignisses ist hierbei der Zeitpunkt der Aktivierung des Eingangssignals am Mikroprozessor, das zur Unterbrechungsanforderung führt (in Ereignis-gesteuerten Systemen) bzw. das bei Auslesen zu einer Ent-

scheidung im Programm führt, dass hier ein Ereignis behandelt werden muss (in Zeit-gesteuerten Systemen).

Definition 3.3:

Die *Wiederholungs- oder Folgezeit* $\boldsymbol{T_F}$ ist die Zeit einer Periode in einem zyklischen oder quasi-zyklischen Design. Im Fall des Zeit-gesteuerten Designs ist dies unmittelbar die (Soll-)Zeit zwischen zwei zeitlich aufeinander folgenden Routinen zur Bestimmung der Ereignisse, im Fall des Ereignis-gesteuerten Designs ist dies die Minimalzeit, die zwischen zwei Ereignissen liegt.

Für das Ereignis-gesteuerte System wurde diese Minimalzeit als gegeben angenommen bzw. im erweiterten System (→ 3.1.6) erzwungen. Im Zeit-gesteuerten System wird sie durch das Systemdesign (Einstellung am Timer) definiert und muss natürlich mit den Erfordernissen am Prozess im Einklang stehen.

Definition 3.4:

Der *maximal akzeptable Jitter* ***Tjt*** ist die maximal tolerable Varianz für die Abweichung vom Soll- bzw. Mittelwert innerhalb eines zyklischen oder quasi-zyklischen Designs. Zur Angabe des Jitters muss die Bezugsgröße mit angegeben werden.

Der Jitter ist eine Zeit bzw. Zeitspanne, die nur im Ausnahmefall zur Beurteilung der Güte benötigt wird. So sind im Normalfall alle Reaktionen zeitlich korrekt, wenn sie im Zeitintervall [0, Tmx] liegen. Bei Messwertaufnahmen hingegen möchte man häufig eine möglichst konstante Differenz der Aufnahmezeitpunkte, um die Messungen nicht künstlich zu verfälschen.

Definition 3.5:

Die *Testzeit* ***TTst*** ist die Zeit einer Periode zum Testen des Außenprozesses, ob ein bestimmtes Signal anliegt und der Prozess hierdurch signalisiert, dass eine Bedienung notwendig ist.

Zwischen den definierten Zeiten existieren folgende Relationen:

$$T_{mx} \leq T_F \tag{3.3}$$

bzw.

$$T_{mx} \leq \overline{T_F} \tag{3.4}$$

Gl. (3.3) muss nicht exakt erfüllt sein, sondern lediglich im zeitlichen Mittelwert über eine fest definierte maximale Anzahl von aufeinander folgenden Durchläufen (3.4). In diesem Fall kann die Einzelmessung schneller wiederholt werden als die eigentliche Reaktion darauf erfolgen kann, so dass im System Puffer zur Zwischenspeicherung eingebaut sein müssen.

Ein Beispiel hierfür ist der Empfang von Paketen aus einem Netzwerk oder über eine Schnittstelle. Wenn in schneller Folge eine maximale Anzahl von Paketen über das Netz gesendet werden (so genannter Burst-Betrieb), so bleibt der Empfän-

ger operabel, wenn er die Pakete zwischenspeichern und bis zum Empfang des nächsten Bursts bearbeiten kann.

Weiterhin muss noch der Zusammenhang zwischen der im Thread eingerichteten Testzeit T_{Tst} und der Wiederholungszeit T_F sowie über Gl. (3.3) indirekt T_{mx} dargestellt werden:

$$T_{Tst} \leq \left\langle \frac{T_F/2 \text{ für Zeit} - \text{Steuerung}}{0 \text{ für Ereignis} - \text{Steuerung}} \right. \tag{3.5}$$

Die Diskrepanz zwischen Ereignis- und Zeitsteuerung kommt ausschließlich durch die zusätzliche Abfrage (in Software) zustande, ob tatsächlich ein Ereignis vorliegt, und dies in einer zeitlichen Abfolge, dass das Ereignis rechtzeitig erkannt wird. Dies bedeutet, dass ein Overhead in Zeit-gesteuerten Systemen entsteht, der die Abfrage enthält und dessen Auftrittsfrequenz im Allgemeinen mindestens $2/T_F$ ist.

Die eigentliche Belastung des Systems durch die Reaktionsberechnung ist in allen Systemen im Übrigen gleich und erfolgt mit der maximalen Frequenz $1/T_F$. Im Sonderfall, dass die Zeit-Steuerung nicht feststellen muss, ob reagiert werden muss, sondern dazu dient, dass agiert wird (z.B. zyklische Messwertaufnahme), entfällt der Overhead im Übrigen, da hier die Zeitsteuerung eigentlich eine Ereignissteuerung darstellt.

> Anmerkung: Im Allgemeinen geht man davon aus, dass sich Zeit- wie Ereignis-gesteuerte Systeme im Rechenzeitbedarf nicht unterscheiden, soweit es um den Nachweis der Echtzeitfähigkeit geht. Konkret haben die Zeit-gesteuerte Systeme haben einen zusätzlichen Overhead, der durch die Signalabfrage entsteht und möglichst minimal gehalten werden soll. Im Extremfall sehr häufiger und auch komplexer Abfragen (→ 4.4.1) kann der zusätzliche Bedarf sehr groß werden, so dass ggf. Sonderlösungen eingeführt werden müssen.

3.2.2 Umsetzung der charakteristischen Zeiten in ein Software-Design

In der Praxis geht man sicher von der Wiederholungszeit T_F eines Ereignisses aus. Dies ist die aus dem Prozess stammende charakteristische Zeit, z.B. die Zeit, die zwischen zwei Messungen liegen darf, um bestimmte Frequenzen noch aufnehmen zu können.

Hieraus definiert man dann die vom System geforderte (und nachzuweisende) maximale Reaktionszeit T_{mx} auf den maximal möglichen Wert $T_{mx} = T_F$. Bei einem Burstbetrieb, siehe Anmerkungen oben, kann auch auf den Mittelwert zurückgegangen werden.

Zwischen dem Jitter T_{jt} und der Testzeit T_{Tst} besteht eine Konkurrenzbeziehung. Ist der Jitter „beliebig", d.h. es liegt keine spezifische Einschränkung der Varianz vor,

wird die Testzeit auf den maximal möglichen Wert $T_F/2$ gesetzt, um den Overhead klein zu halten.

Zwei Ursachen können diesen Maximalwert beschränken:

- Um den Jitter klein zu halten muss die Testzeit verkleinert werden, da automatisch $T_{jt} \geq T_{Tst}$ gilt.
- Die maximal mögliche Reaktionszeit bleibt ja erhalten, unabhängig von dem Zeitpunkt der Feststellung der Reaktionsnotwendigkeit. Die mögliche Reaktionszeit *Tmx* wird im schlechtesten Fall, der für Echtzeitsysteme immer anzunehmen ist, auf den Wert $T_{mx} - T_{Tst}$ eingeschränkt, so dass auch das für eine Verkürzung der Testzeit spricht.

Ausgehend von der Wiederholungszeit T_F der Ereignisse und den zugehörigen größtmöglichen Varianzen *Tjt* müssen also die Software-dominierten Zeiten T_{mx} und T_{Tst} angepasst bzw. ausgewählt werden, ggf. auch durch Performanceerhöhung im Mikroprozessor. Bezüglich der Verkürzung der Testzeit sei allerdings auch auf den Abschnitt 3.1.3 verwiesen: Die dort eingeführte Zeit für die Aktionsperiode, die als ggT der Einzelzeiten definiert wurde, führte ggf. zu einem großen Overhead, ein Effekt, der auch durch Verkürzung der Testzeit T_{Tst} erzeugt werden kann.

3.2.3 Worst-Case-Execution-Time und Worst-Case-Interrupt-Disable-Time

Der nächste Schritt besteht in der Bestimmung der maximalen Reaktionszeiten. Hierzu müssen im ersten Schritt die Worst-Case-Execution-Times (WCET) und im zweiten Schritt dann die Kombination aller Routinen und Reaktionen im System bestimmt werden.

Worst-Case-Execution-Times (WCET)

Die WCET ist diejenige Zeit, die ein Programm oder Programmteil bei ununterbrochener Ausführung maximal benötigt. Diese Zeit kann in Sekunden, aber auch in Prozessortakten angegeben werden. Letztere Angabe ist meist sinnvoll, da der reale Prozessortakt zwischen der Anzahl der Takte und der damit vergangenen Zeit koppelt.

Die Bestimmung der WCET bedeutet, dass die Summe über die Befehle, multipliziert mit der maximalen Ausführungszeit, gebildet werden muss. Zudem muss auch der längste Weg hierfür ausgewählt werden, und dies ist manuell nur für kleine Programme möglich.

Insbesondere muss die Auswertung auf Assembler- bzw. Maschinenbefehlsebene erfolgen, denn in der Hochsprachencodierung kann man die wirkliche Anzahl der Befehle nur sehr schwer erkennen. Allerdings wäre eine Auswertung auf dieser Ebene wesentlich bequemer. Zu dieser Problematik wird noch im Abschnitt 3.3 Stellung genommen.

Weiterhin existieren mehrere Gründe, warum die WCET nicht oder nur sehr unpräzise bestimmt bzw. geschätzt werden kann. Generell gilt es als akzeptabel, wenn die WCET-Werte nicht bestimmt, sondern geschätzt werden, solange die Schätzung sicher ist, d.h. es gilt garantiert WCET(geschätzt) ≥ WCET(real). Die Genauigkeit der geschätzten WCET-Werte gilt bereits als gut, wenn der Fehler < 100% ist, die Schätzung also weniger als den Faktor 2 größer ist als der reale Wert.

Die Schätzung kann aus folgenden Gründen unmöglich bzw. unrealistisch sein:

- Wird im System ein Mikroprozessor/Mikrocontroller mit einem Cache-Speicher eingesetzt, dann ist die Verteilung der Instruktionen und Daten zwischen Cache (schnell) und Hauptspeicher (langsam) unbestimmt. Umfangreiche Studien haben gezeigt, dass es unmöglich ist, einen Mindestbeschleunigungsfaktor zu bestimmen, also bei den aktuellen Cache-Architekturen einen garantierten Beschleunigungswert zu erhalten. Ursache hierfür sind die Ersetzungsstrategien, die einen starken Zufallscharakter haben.

  Abhilfe kann hier nur geschaffen werden, wenn Cache mit anderen, Nicht-Standard-Strategien verwendet wird, oder auf Cache zugunsten eines Scratch-Pad-Memory (schneller Zwischenspeicher, in dem ganze Teilprogramme vom eigentlichen Programm her gesteuert gespeichert und ausgeführt werden) verzichtet wird.

  Ohne diese Abhilfe kann die WCET nur bestimmt werden, indem der Cache als nicht existent angenommen wird. Dies führt dann zu unrealistisch hohen Werten.

```
#include <stdlib.h>
int k, iEnde, iCounter;
int array[1024];

srand();   /* Initialisierung der Pseudo-Zufallszahlen */
iCounter = 0;
iEnde = rand(); /* iEnde besitzt jetzt einen positiven Integerwert */
for( k = 0; k < iEnde; k++ )
{
   iCounter = iCounter + 1;
   if( array[k] == 0 )
      k = 0;
}
```

Bild 3.5 Beispielcode für Schleife, deren Laufzeit nicht zur Übersetzungszeit bestimmt werden kann

- Bestimmte Programmkonstrukte sind nicht zur Bestimmung einer WCET geeignet. Hierzu zählen rekursive Programmierung sowie Schleifen, deren Anzahl der Durchläufe nicht zur Übersetzungszeit bestimmbar ist.

```
int k, iCounter;
int array[1024];

iCounter = 0;
for( k = 0; k < 1024; k++ )
{
   iCounter = iCounter + 1;
   if( array[k] == 0 )
      k = 0;
}
```

Bild 3.6 Beispielcode für unendliche Schleife

```
int factorial( int iNum )
{
   switch( iNum )
   {
      case 0:
         return( 1 );
         break;

      case 1:
         return( 1 );
         break;

      default:
         return( iNum * factorial( iNum – 1 ));
         break;
   }
}
```

Bild 3.7 Beispielcode für rekursive Programmierung (hier: Fakultätsberechnung)

Worst-Case-Interrupt-Disable-Times (WCIDT)

Der Jitter Tjt in einem System bestimmt sozusagen die Genauigkeit der Aktion. Als eine der möglichen Quellen war bereits die Testzeit TTst bestimmt worden, die in einem Zeit-gesteuerten System die Erkennung eines Signals bestimmt.

Die andere, immer vorhandene Quelle für einen Jitter sind unterbrechungslose Zeiten im Programm. Unabhängig davon, ob ein Ereignis-Interrupt oder ein Timer-Interrupt am Prozessor zu behandeln ist, handelt es sich immer um eine Unterbrechung des bisherigen Programmlaufs. Eine solche Unterbrechung ist nicht immer möglich, denn fast immer existieren Programmteile, deren Unterbrechung eine schwere Störung darstellen würde und die ggf. sogar zum Absturz des gesamten Systems führen könnte. Solche Programmabschnitte werden als *atomar* bezeichnet, ein Beispiel hierfür ist in Abschnitt 3.3 gegeben.

Im Softwaredesign wirkt man dieser Gefahr dadurch entgegen, dass gefährdete Programmabschnitte durch die Unterbindung der Interrupts geschützt werden. Die Identifizierung solcher Programmabschnitte ist dabei keineswegs trivial, und hier können nur allgemeine Leitlinien gegeben werden:

- Kandidaten sind zunächst alle Interrupt Service Routinen und alle Funktionen, die im Rahmen einer ISR aufgerufen werden. Häufig wird hier die Strategie verfolgt, dass generell ein (weiterer) Interrupt verboten ist und nur im Ausnahmefall zugelassen wird.
- Weitere Kandidaten sind Abschnitte in Programmen, bei denen auf mehrere globale Variablen, die semantisch zusammenhängend sind, (lesend oder schreibend) zugegriffen wird. Falls diese Variablen durch Interrupt Service Routinen ebenfalls genutzt werden, müssen sie immer gemeinsam geändert werden, weil ansonsten eine nur teilweise geänderte Menge als gültigen Variablensatz gewertet wird.

  Beispiel hierfür ist die Speicherung des aktuellen Zeitpunkts mit Datum und Uhrzeit bis hin zu Millisekunden. Dies wird häufig in mehreren Variablen gespeichert und z.B. durch einen Timer-Interrupt mit zugehöriger ISR ständig wietergesetzt. Wenn nun eine externe Routine diese Zeit benötigt, dann könnte sie beim Stand von 5,999 Sekunden vielleicht 6,999 Sekunden lesen, weil zuerst der Millisekundenstand (999) und dann der Sekundenstand (erst 5, beim Lesen durch Umspringen aber 6) gelesen würde.

Die Worst-Case-Interrupt-Disable-Time (WCIDT) ist nun diejenige Zeit, die die maximale Zeit (auch gemessen in Prozessortakten) darstellt, in der das Programm nicht unterbrechbar ist. Hierzu müssen alle Abschnitte im normalen Programm, die nicht unterbrochen werden dürfen, sowie alle Interrupt Service Routinen, falls diese auch nicht unterbrechbar sind, betrachtet werden.

3.2.4 Nachweis der Echtzeitfähigkeit

Nunmehr besteht zumindest grundsätzlich die Möglichkeit zur Bestimmung der Echtzeitfähigkeit. Das System möge dabei k verschiedene Signale in echtzeitfähiger Weise bearbeiten. Folgende Voraussetzungen seien hierfür angenommen:

- keine ISR kann unterbrochen werden,
- jede ISR behandelt den IRQ vollständig, d.h. die Reaktion ist vollständig hierin beschrieben,
- für jede ISR ist eine eigene Priorität (0 … k–1) gegeben (0 bedeutet dabei die höchste Priorität),
- für jeden IRQ ist eine maximale Frequenz des Auftretens und eine maximale Reaktionszeit gegeben und
- das Hauptprogramm ist jederzeit unterbrechbar.

Folgende Abkürzungen seien ferner für die Bearbeitungs- und Wartezeiten gewählt

Für IRQ(i) sei TF(i) die minimale IRQ-Folge-oder Wiederholungszeit und TMX(i) die maximal zulässige Antwortzeit, TA(i) die Bearbeitungszeit für die i-te Service Routine, alle Zeiten ausgedrückt in Prozessortakten. SP sei diejenige Zeit, die sich als KGV (kleinstes gemeinsames Vielfaches) aller minimalen Folgezeiten TF(i) ergibt, die so genannte *Superperiode*. Ferner sei num(i) = SP/TF(i) die maximale Anzahl der Auftritte pro Zeitintervall SP. Jetzt müssen die Ungleichungen

$$\sum_{i=0}^{k-1} num(i) \cdot TA(i) \leq SP \tag{3.6}$$

$$\forall n \in \{0,...k-1\}: \sum_{i=0}^{n-1} (\left\lceil \frac{num(i)}{num(n)} \right\rceil \cdot TA(i)) + \max_{\substack{j=n+1 \\ ...k-1}} (TA(j)) + TA(n) \leq Tmx(n) \tag{3.7}$$

gelten. (3.6) bedeutet dabei, dass die Summe aller im Zeitintervall der Superperiode SP auftretenden IRQ-Bearbeitungszeiten dieses Intervall nicht überschreiten darf – eine vergleichsweise einfach zu realisierende bzw. nachzuweisende Forderung, die aber nur notwendig (und nicht hinreichend) ist.

(3.7) bedeutet hingegen, dass für alle IRQ-Ebenen (und Prioritäten) die Einhaltung der maximal möglichen Antwortzeit gewährleistet sein muss. Hierzu muss angenommen werden, dass ein niedriger priorisierter IRQ kurz zuvor auftrat und bearbeitet wird, und dass alle höheren IRQs ebenfalls auftreten und bearbeitet werden.

Der mittlere Term bedeutet, dass alle im Intervall zwischen zwei IRQs der Priorität n auftretenden IRQs höherer Priorität mit berücksichtigt werden müssen. Hierzu ist das Verhältnis zwischen Anzahl num(i) und num(n) entscheidend, und zwar der nächst höhere ganzzahlige Wert, denn dieser gibt die Anzahl der möglichen IRQs im Zeitabschnitt an.

Gl. (3.7) ist außerordentlich schwierig (aufwendig) im Nachweis, oder sie bedeutet, dass das System planmäßig überdimensioniert werden muss. Insgesamt sind folgende Vor- und Nachteile für diese Form der Systemauslegung aufzuzählen:

+ Alle zeitkritischen Routinen sind entsprechen priorisiert angelegt, das System kann optimal angepasst werden.
+ Der Nachweis harter Echtzeitfähigkeit ist möglich
- Der Nachweis ist sehr aufwendig
- Ein echtes Scheduling im Sinn einer dynamischen Verteilung kann so nicht erreicht werden.

Der letzte Punkt wird nochmals im Kapitel 4 aufgegriffen, wo ein Design Pattern (Entwurfsmuster) für einen variierten Ansatz diskutiert wird. In diesem Design Pattern kann dann ein Applikations-spezifisches Scheduling durchgeführt werden.

3.3 Kommunikation zwischen Systemteilen

Im bisherigen Systementwurf trat noch keine Kommunikation zwischen (heterogenen) Systemteilen auf, obwohl dies vermutlich schon notwendig ist. Kommunikation wird zwar zumeist mit Netzwerk verbunden, aber innerhalb eines Systems kommunizieren Systemteile (= Threads, Tasks, Prozesse) ebenfalls miteinander.

Datenkommunikation heißt allgemein Austausch von Daten zwischen zwei elektronischen Systemen, wobei zwei unabhängige (Teil-)Programme ebenfalls als je ein elektronisches System gezählt werden sollen. Bezüglich einer Klassifizierung der Kommunikation unterscheidet man in der Art, wie dem Kommunikationspartner die Daten zugänglich gemacht werden:

- Kommunikation per *Shared Memory*
- Kommunikation per Nachrichten (*Message Passing*)

Ferner wird bezüglich des zeitlichen Verhaltens der Kommunikation klassifiziert. Hierbei steht im Vordergrund, wie (und ob) sich die Kommunikationspartner synchronisieren oder nicht (siehe hierzu auch 2.2.3 bez. der grundlegenden Modelle für die Nebenläufigkeit):

- *Nicht-blockierende* Kommunikation
- Synchronisierende, *blockierende* Kommunikation

Hier sei auf die Unterschiede zu 2.2.3 verwiesen. Dort war von (perfekt) synchroner Kommunikation gesprochen worden, dieser Begriff bedeutet, dass die Kommunikationspartner ist Null- bzw. konstanter Zeit miteinander kommuniziert haben. Die blockierende Kommunikation hier wartet auf eine Bestätigung oder ein Ergebnis, und von einer Zeitschranke hierfür ist keine Rede.

3.3.1 Kommunikation per Shared Memory versus Message Passing

Kommunikation mittels *Shared Memory* bedeutet, dass die Kommunikationspartner einen gemeinsamen Speicherbereich und mithin einen gemeinsamen Adressraum besitzen. Dies ist im Allgemeinen nur in lokalen Systemen möglich, wo also der Rechner aus einem oder mehreren Prozessoren, aber eben mit mindestens einem gemeinsamen Speicher besteht.

Die Kommunikation besteht darin, dass ein Teilnehmer in den Datenbereich des anderen hineinschreibt und auf eine vereinbarte Weise das Vorhandensein neuer Daten signalisiert. Vor- und Nachteile dieses Verfahrens:

+ Einfachheit der Kommunikation
+ Die Performance dieser Kommunikation ist optimal, insbesondere große Datenmengen können so in kurzer Zeit übertragen werden
- Das Schreiben in den Adressraum einer anderen Applikation ist gefährlich, weil auch Bereiche unerlaubt überschrieben werden können
- Der Einsatz ist beschränkt auf lokale Systeme, und dabei innerhalb eines Prozesses (falls das Betriebssystem Multiprocessing unterstützt), weil 2 verschiedene Prozesse per definitionem unterschiedlichen Adressräume haben

Kommunikation per *Message Passing* bedeutet, dass man einen ausgezeichneten Kommunikationskanal hat. Dieser Kommunikationskanal kann vom Betriebssystem geschaffen, es kann sich um ein Netzwerk handeln usw. Vor- und Nachteile sind:

+ Kommunikation auf „beliebige“ Entfernung
+ Gesichertes Modell (gegenüber Fehlhandlungen)
- Schwierigkeiten bei hohen Performance-Anforderungen und bei geforderter sicherer und/oder Echtzeitübertragung
- Komplexere Entwurfsmethodik

3.3.2 Blockierende und nicht-blockierende Kommunikation

Abgesehen von Entfernungs- und Performancefragen, die im Abschnitt 3.3.1 mitdiskutiert wurden, existieren weitere Anforderungen zur Kommunikation, die teilweise in Konkurrenz stehen:

- Sicherstellung der Kommunikation (mit Benachrichtigung bei Fehlschlag oder Fehlern)
- Beeinflussung des eigenen Laufzeitverhaltens, etwa durch Warten
- Garantie einer maximalen Laufzeit (mit Benachrichtigung bei Überschreiten)

Die Garantie einer maximalen Laufzeit wird in erster Linie auf das Netzwerk selbst abgebildet, da im umgekehrten Fall ein nicht-echtzeitfähiges Netzwerk nicht ausgleichbar ist. Die Eigenschaft einer echtzeitfähigen Kommunikation ist in Netzwerken aber nur mit vergleichsweise hohem Aufwand realisierbar:

- Als wichtigste Voraussetzung gilt, dass die Ressource Netzwerk (pro Zeiteinheit ist nur eine Übertragung möglich) deterministisch zugeteilt wird. Als besonders geeignetes Verfahren erweist sich hier die zeitlich gesteuerte Zuteilung (time-triggered), wo alle Netzteilnehmer eine einheitliche Zeit führen und Zeittabellen besitzen, wann sie selbst und wann die anderen senden (dürfen/müssen) (siehe auch 4.4.2 und 4.4.3)
- Die Steuerung per Zeittabellen bei gleichzeitigem ständigen Zeitabgleich ermöglicht sogar eine Ausfallerkennung, allerdings ist der Durchsatz hier suboptimal.

Aus Sicht der Applikation ist die Einführung eines echtzeitfähigen Netzwerks notwendige, aber nicht hinreichende Voraussetzung für die Echtzeitfähigkeit der Gesamtapplikation. Außer der eigenen Echtzeitfähigkeit aller Teilkomponenten muss auch die Kommunikation in der Applikation nicht-blockierend ausgelegt sein, wie an folgendem (lokalen) Beispiel ersichtlich ist:

```
unsigned char semaMess = 0;
unsigned int globalMesswert;

...
main()
{
  unsigned int messwert;
  ...
  while(1)
  {
    if( semaMess != 0 )
    { /* Neuer Wert vorliegend? */
      /* Atomare Operation */
      #asm( cli );
      semaMess = 0;
      messwert = globalMesswert;
      #asm( sei );
      /* Ende der atomaren Operation */
      ...
    }
  }
}
```

a)

```
interrupt [TIMER] void timer_comp_isr(void)
{

  /* Die beiden Operationen sind wieder
    atomar */
  globalMesswert = ADC_OUT;
  semaMess = 1;

  ...
}
```

b)

Bild 3.8 Nicht-blockierende Kommunikation zwischen Main- (a) und Interrupt-Routine (b)

Bild 3.8 zeigt zu diesem Zweck eine nicht-blockierende Kommunikation zwischen einem *Producer* (hier: Interrupt-Service-Routine, die Messwerte aus einem AD-Umsetzer ausliest und speichert) und einem *Consumer* (Teil der Hautapplikation). Der Producer wartet nicht darauf, ob ein Wert schon gelesen und verarbeitet wurde (Kennzeichen: semaMess == 0), sondern schreibt den neuen Wert in die vorgesehene Hauptspeicherstelle, auch auf die Gefahr eines Werteverlusts.

Auch der Consumer wartet nicht auf einen neuen Wert, sondern beginnt seine Auswertung nur dann, wenn ein neuer Wert vorliegt, ansonsten könnte etwas anderes durchgeführt werden. Diese Form der nicht-blockierenden Kommunikation ist wichtig für Echtzeitsysteme, denn die Blockierung könnte unabsehbare Auswirkungen auf das Timing haben.

Anmerkung: Das Schreiben einer nicht-blockierenden Kommunikation ist oftmals nicht trivial, weil die Aktion, die auf der Kommunikation beruht, ja grundsätzlich ausgeführt werden soll. Hierzu sei ein Beispiel gegeben:

Angenommen, ein Satz von 16 Daten soll in das EEPROM eines Mikrocontrollers geschrieben werden, weil die Werte ein Abschalten der Spannung überdauern sollen. Das Schreiben in das EEPROM wird von modernen C-Compilern mithilfe von Laufzeitbibliotheken unterstützt, aus Sicht der Anwendungsprogrammierung ist dies eine einfache Sache:

```
eeprom int eepIBasicConfig[16];
int k, iConfig[16];
…
for( k = 0; k < 16; k++ )
   eepIBasicConfig[k] = iConfig[k];
```

Das Gefährliche an dieser Codesequenz ist, dass hinter der scheinbar einfachen Wertzuweisung an eepIBasicConfig[] eine ganze Laufzeitroutine mit blockierender Kommunikation steckt, was so nicht ersichtlich ist. Meistens wird der erste Wert sofort geschrieben (weil in der Hardware ein Pufferplatz vorhanden ist, ab dem zweiten Wert wartet man dann auf die Fertigstellung des Schreibens des Vorgängerwerts – was durchaus einige Millisekunden dauern kann.

Die Kunst der nicht-blockierenden Kommunikation will es nun, dass man nur hineinschreibt, wenn dies sofort möglich ist (weil der Puffer frei ist), ansonsten sich merkt, dass noch etwas zu schreiben ist, und zum weiteren Programm zurückkehrt. Hierbei muss dann gewährleistet sein, dass das Schreiben etwa durch zyklisches Design irgendwann fertig gestellt wird, und dass es keine Seiteneffekte gibt.

Die Routine in Bild 3.8 beinhaltet noch einen atomaren Teil (→ 3.2.3): Durch die Unterbindung aller Interrupt Requests in der Hauptroutine (zwischen #asm(cli) und #asm(sei), Befehle, die das Interrupt Enable Föag löschen bzw. setzen) werden die darin enthalten C-Anweisungen ununterbrechbar und somit immer zusammenhängend durchgeführt.

4 Design-Pattern für Echtzeitsysteme, basierend auf Mikrocontroller

4.1 Dynamischer Ansatz zum Multitasking

Die in Kapitel 3 diskutierten Verfahren beruhen auf der Idee, die zeitkritischen Teile in eine Unterbrechungsroutine einzufügen und den Rest der Zeit die relativ zeitunkritischen Teilaufgaben zu rechnen. Es fehlt jedoch noch die Zusammenfassung dieser Teile in einem Programm bzw. ein Design-Pattern für das komplette Systemdesign.

Das hier vorgestellte Designverfahren beruht auf drei Schritten:

- Klassifizierung der Teilaufgaben
- Implementierung der Einzelteile
- Zusammenfassung zum Gesamtprogramm

Ein Hauptaugenmerk muss dabei auf die Kommunikation zwischen den Threads ($\rightarrow$ 3.3) gelegt werden.

4.1.1 Klassifizierung der Teilaufgaben

Das hier dargestellte Designverfahren beruht darauf, die einzelnen Teilaufgaben (diese werden hier immer als Thread bezeichnet) zu klassifizieren, ihren gewünschten Eigenschaften nach zu implementieren und das System dann zu integrieren. Die folgende Klassifizierung ist notwendig, da insbesondere im Zeitbereich verschiedene Randbedingungen für die einzelnen Klassen angenommen werden müssen.

- *Streng zyklisch ablaufende Threads*: Fester Bestandteil dieser Teilaufgaben sind exakte Zeitabstände, in denen diese Threads zumindest gestartet werden und generell auch komplett ablaufen müssen, um der Spezifikation zu genügen. Beispiele hierfür sind Messwertaufnahmen oder die Bedienung von asynchronen Schnittstellen zur Datenkommunikation.
- *Ereignis-gesteuerte Threads*: Das Starten bzw. Wecken einer Threadmit dieser Charakterisierung ist an ein externes Ereignis gebunden, meist in Form eines Interrupt-Requests. Der Startzeitpunkt ist somit nicht zur Compilezeit bestimmbar, so dass diese Threads störend auf den zeitlichen Gesamtablauf wirken können. Typische Vertreter dieser Klasse sind der Empfang von Nachrichten via Netzwerk bzw. die Reaktion darauf oder Schalter in der Applikation, die besondere Zustände signalisieren (etwa ”Not-Aus“).
- *Generelle Tasks mit Zeitbindung*: Die dritte Klasse beschreibt alle Threads in dem System, die zwar keine scharfen Zeitbedingungen enthalten, im Ganzen

jedoch Zeitschranken einhalten müssen. Hiermit sind Threads beschrieben, die beispielsweise Auswertungen von Messwerten vornehmen. Während die einzelne Auswertung ausnahmsweise über einen Messwertzyklus hinaus dauern darf, muss insgesamt die mittlere Auswertezeit eingehalten werden.

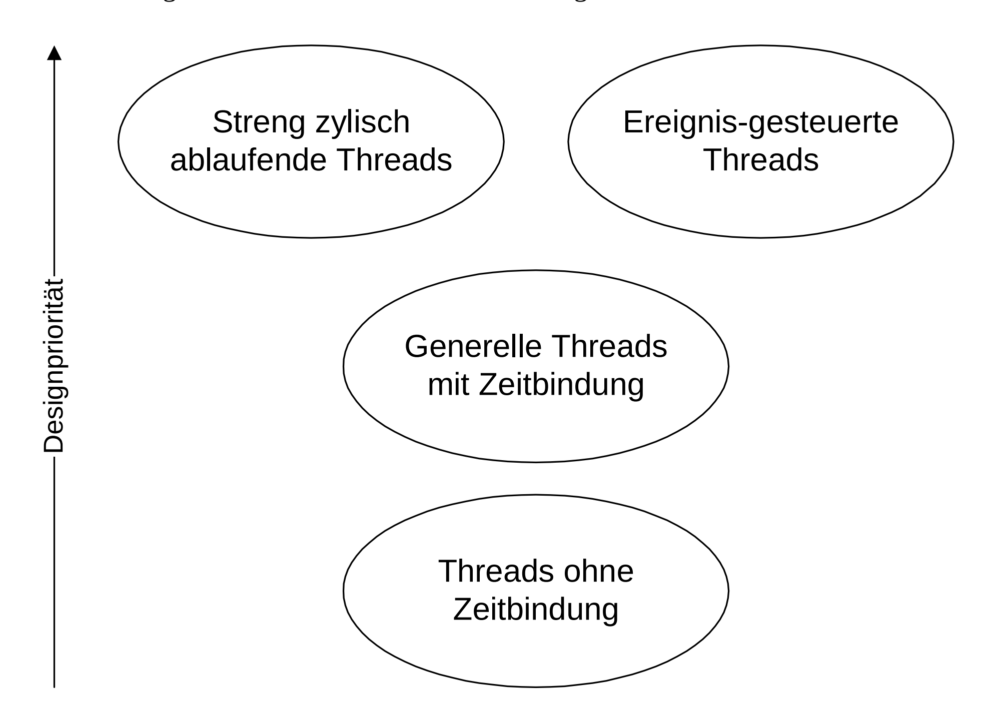


Bild 4.1 Threadklassen und Designprioritäten

Diese drei Grundklassen zeitabhängiger Teilaufgaben stellen das Grundgerüst zum Systemdesign dar. Die erste Aufgabe des Systemdesigners besteht darin, alle in der Beschreibung vorkommenden Aufgaben in dieses Grundgerüst einzuteilen, mit allen dabei auftretenden Schwierigkeiten.

Generell gilt, dass eine Teilaufgabe in eine "höhere Klasse" integriert werden kann. So kann ein Thread, die überhaupt keine Zeitbindung besitzt – dies dürfte in der Praxis nicht häufig vorkommen, ein Kontrollthread wäre aber hier möglich – natürlich in die Klasse der generellen Threads mit Zeitbindung sortiert werden. Diese Threadklasse ist in Bild 4.1 dargestellt, wurde jedoch nicht in die Klassifizierung aufgenommen, da sie irrelevant für das hier dargestellte Designprinzip ist.

Streng zyklisch ablaufende Threads und Ereignis-gesteuerte Threads sind in ihrer Designpriorität in etwa gleichzusetzen (→ Bild 4.1). In der Praxis kann die Implementierung auch sehr ähnlich sein, indem die zyklischen Threads in Interrupt-Service-Routinen (ISR) mit Timersteuerung und die Ereignis-gesteuerten Threads

in anderen ISRs behandelt werden. Die Unterscheidung soll dennoch aufrecht erhalten bleiben, da zwischen beiden Implementierungen ein fundamentaler Unterschied existiert.

4.1.2 Lösungsansätze für die verschiedenen Aufgabenklassen

Im nächsten Schritt des Designverfahren werden die Mitglieder der einzelnen Klassen zunächst getrennt voneinander implementiert und die maximale Ausführungszeit jeweils berechnet. In erster Näherung werden dafür die WCET der einzelnen Teilaufgaben als voneinander unabhängig angenommen. Um dies wirklich zu erreichen, muss auf ein blockierendes Warten bei Kommunikation zwischen den Threads unbedingt verzichtet werden, denn dies kann zu großen Problemen bei der Bestimmung der WCET bis hin zur Unmöglichkeit führen (→ 3.3.2). Diese Forderung führt zu einem sicheren Design, da sich Abhängigkeiten etwa in der Form, dass, falls Thread 1 den maximalen Pfad durchläuft, Thread 2 garantiert einen kleineren Pfad als seinen maximalen wählt, nur positiv auf die WCET des Gesamtsystems auswirken können.

Bild 4.2 zeigt den gesamten Designprozess (ohne Entscheidungen bzw. Rückwirkungen). Tatsächlich sind in seinem Verlauf einige Abstimmungen und Entscheidungen notwendig, insbesondere in dem grau schattierten Teil der Implementierung zweier ISRs mit gegenseitiger Beeinflussung.

Das Zusammenfügen der einzelnen Applikationsteile, bestehend aus generellen Threads, Timer-ISRs und ggf. Event-ISRs, beinhaltet die Organisation der Kommunikation zwischen den einzelnen Teilen sowie die Abstimmung des Zeitverhaltens. Als Kommunikation zwischen diesen Threads ist ein nicht-blockierendes Semaphoren/Mailbox-System ideal: Semaphore, die seitens eines Threads beschrieben und seitens der anderen gelesen und damit wieder gelöscht werden können, zeigen den Kommunikationsbedarf an, während die eigentliche Meldung in einer Mailbox hinterlegt wird.

Blockieren kann durch eine asynchrone Kommunikation wirksam vermieden werden: Threads warten nicht auf den Empfang bzw. Antwort, sie senden einfach (via Semaphor/Mailbox). Auch die Abfrage von empfangenen Sendungen erfolgt dann nicht-blockierend. Dies lässt sich durch einfache Methoden implementieren, wie am folgenden Beispiel ersichtlich ist.

Entscheidend ist die Einführung einer globalen Variablen zur Steuerung der Kommunikation (semaMess). Trägt diese den Wert 0, so liegt kein Messwert vor, und die Hauptroutine, die in eine Endlosschleife eingepackt ist, läuft weiter. Ansonsten wird der Messwert lokal kopiert und die Semaphore semaMess wieder zurückgesetzt, um für den nächsten Schleifendurchlauf einen korrekten Wert zu haben.

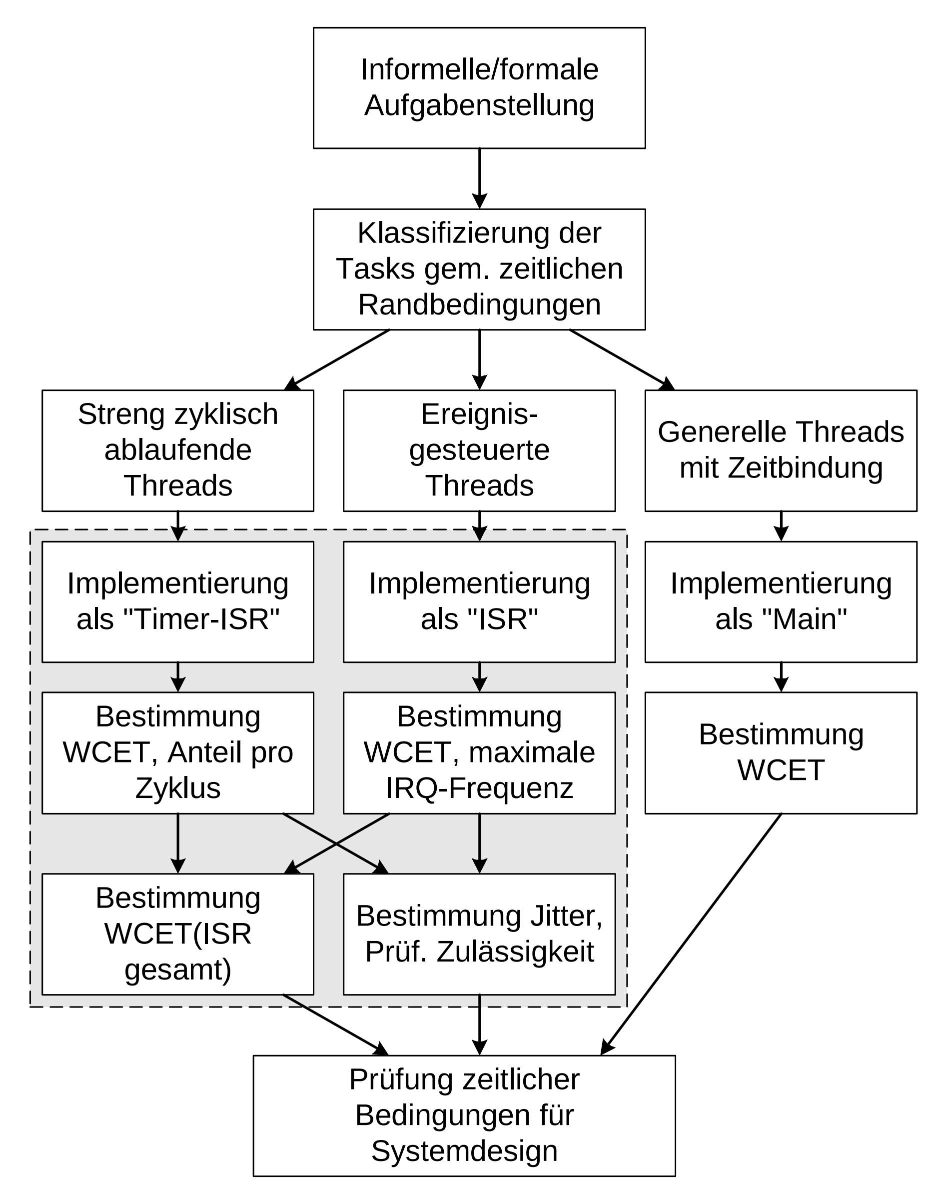


Bild 4.2 Gesamtablauf Systemdesign

Die Interrupt-Service-Routine, hier für einen Timer beschrieben, setzt zugleich den Messwert und die Kommunikationsvariable. Es wird im Codebeispiel davon ausgegangen, dass die ISR nicht unterbrechbar ist, so dass also die beiden Operationen immer hintereinander ausgeführt werden. Um dies im Hauptprogramm zu errei-

chen, muss kurzzeitig der Interrupt gesperrt werden (durch die beiden Assemblerbefehle cli (Clear Interrupt Enable) und sei (Set Interrupt Enable)). Hierdurch erreicht man im Code die geforderte Atomarität, die für einen ungestörten Ablauf zwingend erforderlich ist (→ 3.2.3, 3.3.2).

Die zeitliche Abstimmung der einzelnen Threads ist wesentlich aufwendiger und muss folgende Überlegungen einschließen:

- Wie beeinflussen ununterbrechbare Teile in dem generellen Thread bzw. einer ISR die Latenzzeiten der Interrupts? Die Beantwortung dieser Frage ist insbesondere für die zyklischen Threads mit strenger Zeitbindung wichtig, da man hier davon ausgehen muss, dass Jitter nur in sehr geringem Umfang erlaubt ist.

  Die praktische Ausführung sieht so aus, dass tatsächlich die entsprechenden Befehle im Maschinencode (”sei“, ”cli“) gesucht und die WCETs der ununterbrechbaren Zwischenräume bestimmt werden. Diese Zeiten können mit WCIDT (Worst-Case Interrupt Disable Time) bezeichnet werden und sollten auf das absolute Minimum beschränkt sein, z.B. auf ”atomare“ Aktionen zur Kommunikation. Die Bestimmung hierzu muss am endgültigen Maschinencode erfolgen, um eingebundene Laufzeitroutinen zu erfassen.

- Timer-ISR und Event-ISR stehen in Konkurrenzbeziehung, was die Zuteilung der Rechenzeit betrifft. Grundsätzlich sollte der strengen Zeitbindung der Vorrang gegeben werden, und die Routinen hierfür sind auch Kandidaten für eine Ununterbrechbarkeit. Dies allerdings bedeutet die Erhöhung der Latenzzeit für die Event-ISR, was für den Einzelfall zu prüfen ist.

  Eine Ausnahme bildet der Fall, dass die Event-ISR sehr hoch priorisiert werden muss, weil bei Auftreten ein sicherer Zustand zu erreichen ist. Dieses Ereignis muss sofort behandelt werden, so dass die Timer-ISR in diesem Fall unterbrechbar sein sollte.

Nach dem Zusammenfügen der einzelnen Teile und der Abstimmung der zeitlichen Randbedingungen kann dann die korrekte Funktionsweise des gesamten Systems nachgewiesen werden. Hierzu wird ein Zeitraum betrachtet, in dem ein gesamter Zyklus ablaufen kann. Insbesondere muss die generelle Task die Berechnung beenden können. In diesem Zeitabschnitt darf die Summe der WCETs, multipliziert mit den entsprechenden Auftrittshäufigkeiten, die Gesamtrechenzeit nicht überschreiten.

Für die Latenzzeiten gelten die gesonderten, oben beschriebenen Bedingungen.

4.2 Design-Pattern: Software Events

Der im vorigen Abschnitt beschriebene Mechanismus scheint darauf hinaus zu laufen, alle zeitkritischen Aktivitäten komplett in ISR zu halten und den Rest im Rahmen einer großen `while(1)`-Schleife im Hauptprogramm zu halten. Das

wäre insgesamt nicht besondere wartungsfreundlich, abgesehen davon, dass die ISR stark überlastet bzw. überfrachtet wären.

Dem Betriebssystem-losen Designansatz aus Abschnitt 4.1 wird nämlich gerne nachgesagt, dass er nur zu einer großen "`main()`-Schleife" ohne jegliche Struktur führt. Das Ergebnis wäre in der Tat schlecht, denn die Pflegbarkeit eines Programms steht und fällt mit der im Programm eingebauten Struktur.

Dieses scheinbare Manko kann aber mit einfachen Designmitteln vermieden werden, indem ein kleiner Teil des Betriebssystems mit sehr einfachen Mitteln nachgebildet wird: Der Scheduler.

Dem *Scheduling* obliegt die Aufgabe, das aktive Rechnen zwischen verschiedenen Teilen des Softwaresystems, hier als Tasks bezeichnet, hin- und herzuschalten. Folgende Struktur sei dem Gesamtsystem zugrunde gelegt:

- Es existieren verschiedene Quellen für Reaktionsanforderungen: Externe Hardware wie z.B. AD-Umsetzer, Netzwerkanschluss oder Sensoren werden mit internen Quellen wie Timer im System gemischt.
- Auf jede Reaktionsanforderung, im Folgenden *Event* genannt, wird auf gesonderte Weise reagiert, d.h. zu jedem Event gehört eine relativ komplexe Reaktionsroutine.

Für ein derartiges System kann die Kopplung zwischen Events verschiedener Quellen und den zugehörigen Reaktionsquellen durch folgendes Designpattern (Entwurfsmuster), das mit *Software Events* bezeichnet wird, erreicht werden:

4.2.1 1. Stufe: Vom Hardware- zum Softwareereignis

Es sei angenommen, dass jedes Ereignis des Außenprozesses in einer Interrupt Service Routine (ISR) behandelt wird. Diese Annahme stützt sich darauf, dass entweder das Ereignis zu einem Interrupt Request selbst führt und dieser dann die ISR aufruft (Ereignis-gesteuertes Design), oder dass im Rahmen der Behandlung eines Timer-Interrupts die Abfrage externer Quellen erfolgt (Zeit-gesteuertes Design).

In beiden Fällen wird in der betreffenden Interrupt Service Routine aus dem Hardware-Ereignis heraus ein Software-Ereignis erzeugt und in einer Ereignisqueue gespeichert. Der entsprechende Code könnte also wie in Bild 4.3 dargestellt aussehen:

Diese Routine schafft die Vereinheitlichung aller Ereignisse zur zentralen Bearbeitung in Software und führt zugleich eine (allerdings nicht perfekte) zeitliche Ordnung ein. Die hierbei genutzte Routine `vStoreNewEvent()` verwaltet dabei die Ereignisqueue, die im einfachsten Fall als FIFO-Buffer geführt und als Ringpuffer verwaltet werden wird. Die Verwaltung gelingt besonders einfach, wenn man die Ereigniseinträge als Struktur definiert:

```
void interrupt vISR()
{
    int iISRRegister;
    iISRRegister = iGetISRRegister();
    if( ISR_FLAG1 == (iISRRegister & ISR_FLAG1) )
    {
         iStoreNewEvent( EVENT_1, 0 );
    }    /* Pseudodatum 0  */
    else if (…)
         …
}
```

Bild 4.3 Rahmen für ISR-Routine zur Umsetzung von Hardware- in Software-Events

```
#define NUM_OF_EVENTS   16
struct stcEvent {
                  int iEventType;
                  int iEventData;
                  };
struct stcEvent stcEventList[NUM_OF_EVENTS];
int iEventWrite = 0, iEventRead = 0;
```

Bild 4.4 Definitionen für die Ereignisliste

Die Funktionen zur Verwaltung der Liste haben dann folgende in Bild 4.5 dargestellte Rahmengestalt:

Die Funktionen zur Verwaltung sind recht einfach gehalten. `iStoreEvent()` liefert einen Rückgabewert, der das erfolgreiche Speichern bzw. den Misserfolg signalisiert, in der ISR hier allerdings nicht ausgewertet wird.

Diese Form der Ereignisübermittlung kann beliebig erweitert werden; so sind Subtypen, weitere Daten, Uhrzeit etc. möglich und können direkt eingetragen werden.

```
int iStoreNewEvent( int iEventType, int iEventData )
{
    if( stcEventList[iEventWrite].iEventType != EVENT_NO_TYPE )
    {
        return( 0 );
    }   /* Rückgabewert 0 bedeutet, dass die Liste voll ist  */

    else
    {
        stcEventList[iEventWrite].iEventType = iEventType;
        stcEventList[iEventWrite].iEventData = iEventData;

        iEventWrite++;
        if( iEventWrite >= NUM_OF_EVENTS )
            iEventWrite = 0;

        return( 1 );
    }   /* Rückgabewert 1 bedeutet, dass das Ereignis gespeichert wurde  */
}

/********************************************************************/
void vGetNextEvent( struct stcEvent *stclTemp )
{
    if( stcEventList[iEventRead].ui8EventType != EVENT_NO_TYPE )
    {
        *stclTemp = stcEventList[iEventRead];
        stcEventList[iEventRead].iEventType = EVENT_NO_TYPE;
            /* Freigeben des Speichers  */

        iEventRead++;
        if( iEventRead >= NUM_OF_EVENTS )
            iEventRead = 0;
    }

    else
    {
        stclTemp->iEventType = EVENT_NO_TYPE;
    }   /* Bedeutet, dass kein Ereignis vorhanden ist  */
}
```

Bild 4.5 Funktionen zur Verwaltung der Ereignisliste

4.2.2 2. Stufe: Bearbeitung der Software-Ereignisliste

Im Hauptprogramm kann die Software-Ereignisliste innerhalb einer Endlosschleife abgearbeitet werden, da davon ausgegangen wird, dass *alle* Ereignisse und damit *alle* zu bearbeitenden Aufgaben in der Liste stehen. Die Endlosschleife hat damit folgende in Bild 4.6 dargestellte Gestalt:

Die darin verwendete Funktion `vExecuteEventList()` stellt den Scheduler dieses Systems dar: In der hier angedeuteten einfachen Form werden alle Ereignisse und deren damit zusammenhängenden Bearbeitungsroutinen der zeitlichen Reihenfolge nach bearbeitet. Hier lassen sich aber auch komplett andere Strategien mit Prioritäten usw. implementieren, so dass die zentrale Entscheidung, was wann bearbeitet wird, hier gefällt werden kann.

```
main()
{
...
    while( 1 )
    {
        vExecuteEventList();
    }
}


void vExecuteEventList( void )
{
    struct stcEvent stclTemp;

    vGetNextEvent( &stclTemp );

    switch( stclTemp.iEventType )
    {
        case 0:
            Rechne_fuer_Case0();
            break;

        case 1:
            Rechne_fuer_Case1();
            break;

        default:
            break;
    }
}
```

Bild 4.6 Bearbeitung der Ereignisliste

4.2.3 Kritische Würdigung dieses Design Pattern

Die Einfachheit dieses Multitasking-Ansatzes darf nicht über die Schwächen hinwegtäuschen. Zu den Schwächen zählen:

- Der Ansatz arbeitet nur kooperativ: Die im Scheduler aufgerufenen Funktionen müssen wieder zurückkehren, terminieren, um weitere Ereignisse bearbeitet zu lassen. Präemptives Scheduling, wie es aus Betriebssystemen bekannt ist, lässt sich so nicht realisieren.
- Grundsätzlich besteht sogar die Möglichkeit, die Ereignisliste selbst zu blockieren, wenn sie mit gerade nicht bearbeitbaren Ereignissen komplett gefüllt wird. Es besteht die Gefahr eines *Deadlocks*, wenn alle Ereignisse gegenseitig aufeinander warten und kein Platz mehr in der Eventqueue vorhanden ist.
- Der Fall, dass Ereignisse vorübergehend nicht mehr gespeichert werden können, ist hier nicht vorgesehen bzw. wird nicht behandelt. Dies gehört aber zu einem sicherheitskritischen System, insbesondere, wenn keine Ereignisse verloren gehen dürfen. Hier ist die Größe des Ereignispuffers in Abhängigkeit von

der Bearbeitungs- bzw. Leerungsgeschwindigkeit und der maximalen Erzeugungsrate entscheidend.

4.3 Komplett statischer Ansatz durch Mischung der Tasks

Ein in [Dea04] dargestellter Ansatz verzichtet sowohl auf ein Scheduling durch ein Betriebssystem als auch auf die Einbindung von Interrupt Service Routinen. Kurz gesagt besteht die Methode darin, den zeitkritischen Teil derart mit dem unkritischeren Teil zu mischen, dass sich – zur Übersetzungszeit berechnet – ein richtiges Zeitgefüge in der Applikation einstellt.

Die Idee wird als ”Software Thread Integration (STI)“ bezeichnet und ist natürlich bestechend einfach. Prinzipiell kann jeder Softwareentwickler dies durchführen, indem – nach sorgfältiger Analyse – die Sourcecodes der einzelnen Threads gemischt werden.

Das Problem ist, dass zugleich ein zyklusgenaues Ausführen des Programms gefordert wird, wenn harte Echtzeitbedingungen einzuhalten sind. Zyklusgenauigkeit ist aber derzeit nur unter mehreren Bedingungen erreichbar:

- Die Anzahl der Ausführungstakte im Mikrocontroller muss zur Übersetzungszeit bestimmbar sein. Hiermit scheiden bisherige Cache-Konzepte aus, denn sie ermöglichen nur statistische, nicht deterministische Aussagen.
- Alternativpfade (if – else) müssen die gleiche Anzahl an Taktschritten aufweisen.
- Die Bestimmung der Anzahl der Ausführungstakte (WCET) muss in der Programmiersprache möglich sein.

Der erste Punkt ist fast automatisch dadurch erfüllt, dass sich diese Methode auf kleine Mikrocontroller (”low-MIPS world“) bezieht. Diese Mikrocontroller besitzen keinen Cache, weil sie zumeist auch nur mit geringen Taktraten versehen sind (etwa 20 MHz) und weil der Cache-Speicher sehr teuer wäre.

Punkt 3, die Bestimmung der Anzahl der Ausführungstakte im Rahmen des Codes, ist auf Ebene einer Hochsprache zurzeit nicht möglich. Hier muss man auf Assembler ausweichen, was mit erheblichen Problemen verbunden ist. Hierunter fällt auch zugleich Punkt 2, denn die eventuelle Auffüllung von schnelleren Pfaden mit ’NOP’-Befehlen (no operation) zwecks Angleichung kann wiederum nur auf Assemblerebene erfolgen.

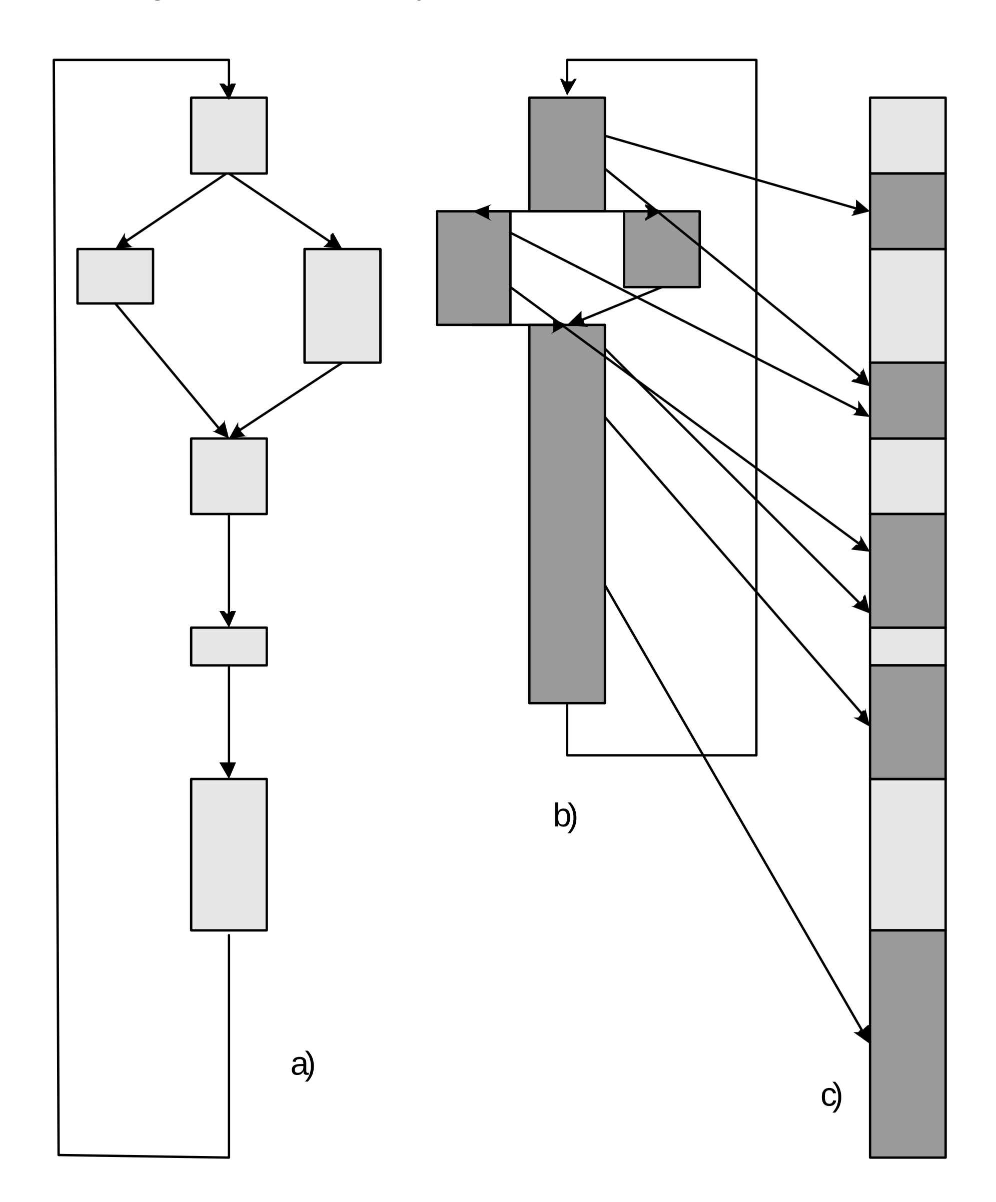


Bild 4.7 Mischung zweier Threads zwecks Software Thread Integration
a) Primärthread, zeitkritisch b) Sekundärthread, zeitunkritisch c) Thread Integration

Folgerichtig bemüht sich der Autor in [Dea04] um eine neue Compilertechnologie, die nach Übersetzung in Assemblercode diesen analysiert, die unterschiedlichen Wege in ihrer Ausführungszeit angleicht und schließlich den Code mischt.

Nach Bestimmung der Ausführungszeiten wird (→ Bild 4.7) der zeitkritische Code zyklusgenau in den auszuführenden Softwarethread eingefügt. Hier ist auch offensichtlich, dass alle Zweige einer Verzweigung die gleiche Laufzeit aufweisen müssen, weil ansonsten von dem Zeitschema abgewichen wird. Die Lücken werden dann durch zeitlich unkritische Teile aufgefüllt.

Dieses Verfahren wirft eine Reihe von Fragen auf, die Compilertechnologie betreffend. Möglich ist es grundsätzlich, wenn die Worst-Case Execution Time (WCET) gleich der Best-Case Execution Time (BCET) ist. Die in [Dea04] dargestellten Methoden, um den Code zu mischen, sind dann von der Güte der WCET-Bestimmung und den Möglichkeiten des Compilers, möglichst einfache Threadwechsel einzubauen, abhängig. Der Gewinn an Performance, verglichen mit einem normalen Scheduling, ist allerdings beträchtlich, er wird mit bis zum Faktor 2 an Performance quantifiziert.

4.4 Co-Design Ansatz: Partitionierung in PLD- und Prozessoranteile

Implizit wurde bei allen bisherigen Modellen zur Echtzeitfähigkeit vorausgesetzt, dass die charakteristischen Zeiten wie Reaktionszeit, Antwortzeit usw. wesentlich größer sind als die Zeit, die ein Prozessor zur Bearbeitung eines Befehls benötigt. Dies muss vorausgesetzt werden, weil der Prozessor in der zeitsequenziellen Ausführungsdimension arbeitet: Er benötigt einfach viele Befehle, um ein Programm zu bearbeiten, und jeder Befehl benötigt etwas Zeit (ca. 1 Takt).

Bild 4.8 a) zeigt nun ein Beispiel für eine relativ einfache Ansteuerung eines AD-Wandlungsvorgangs. Diese Routine ist als Interrupt-Serviceroutine ausgelegt. Angestoßen beispielsweise durch einen zyklischen Timer-IRQ wird der AD-Wandler auf einen neuen Wert abgefragt, und dieser neue Wert wird mit gegebenen Grenzen verglichen. Bleibt der Wert in den Grenzen, passiert nichts, ansonsten wird die out_of_range()-Routine aufgerufen.

Bild 4.8 b) zeigt nun die Assemblerübersetzung dieser Routine für einen hypothetischen Prozessor. In dem Fall, dass kein Grenzwert verletzt wird, benötigt die Routine 14 Instruktionen, bei 1 Instruktion/Takt (RISC-Verhältnis) also 14 Takte oder 140 ns bei 100 MHz.

Dies erscheint als nicht besonders viel, aber bei einer AD-Wandlungsrate von 1 MSPS (Mega-Samples-per-Second) sind das 14% der gesamten Rechenkapazität des Prozessors. Hieraus lässt sich schon ein ungefähres Maß dafür ableiten, wann die Behandlung von Ereignissen in nicht-exklusiver Hardware schwierig bis unmöglich wird. Folgende Kriterien können angegeben werden:

```
int *p_adc, adc_value, upper_limit, lower_limit;
...
void interrupt read_and_compare_ADC()
{
    adc_value = *p_adc;              // Access to AD converter
    if( adc_value > upper_limit  ||  adc_value < lower_limit )
    {
        out_of_range();              // call to exception routine
    }
}
```

Bild 4.8 a) C-Sourcecode für ISR zur AD-Konvertierung mit Grenzwertvergleich

```
TIMER:  push  r0            ;
        push  r1            ;
        push  r2            ;
        mov   r0, ADC       ; Lesen des AD-Werts, zugleich Neustart der Wandlung
        mov   r1, UP_LIMIT  ; Speicherstelle für oberes Limit
        mov   r2, DN_LIMIT  ; dito, untere Grenze
        cmp   r0, r1        ; Grenzen werden verglichen
        bgt   T1            ; Überschreitung, spezielle Routine!
        cmp   r0, r2        ;
        bge   T2            ; Keine Unterschreitung, dann Sprung
T1:     call  OUT_OF_RANGE;
T2:     pop   r2            ;
        pop   r1            ;
        pop   r0            ;
        reti                ; Beenden der Serviceroutine
```

Bild 4.8 b) Assemblerübersetzung

- Wiederholungsfrequenz > 1/100 .. 1/1000 * Prozessorfrequenz
- Geforderter Jitter (Abweichung des Starts der Reaktionsroutine) < 10 .. 1000 Instruktionszeiten
- Bearbeitungszeit einer ISR > 10% Gesamtbearbeitungszeit

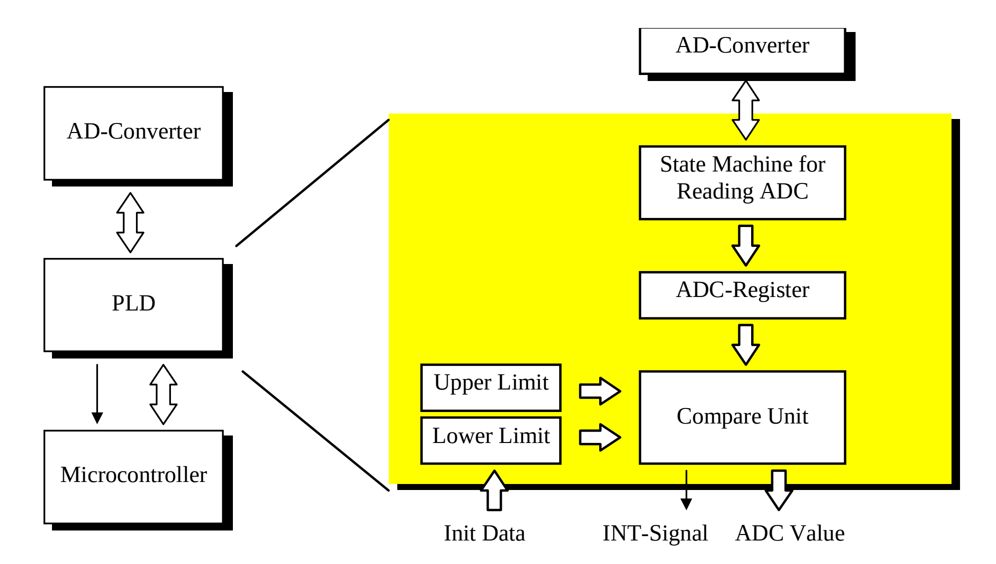


Bild 4.9 Implementierung der AD-ISR in PLD

Die angegebenen Grenzen sind unscharf, sie sollen lediglich zeigen, dass man bei keinem noch so gut ausgelegten Prozessor-basierten System beliebig kleine und scharfe Reaktionszeiten erwarten kann. Für diesen Fall bietet sich eine Partitionierung des Systems an, die besonders kritischen Teile können in exklusiver Hardware untergebracht werden.

Aktuell sind hierfür Kombinationen aus Prozessor und PLD am Markt erhältlich. Beide sind programmierbar, wenn auch auf vollkommen verschiedene Weisen, so dass der Entwickler in das Gebiet des Co-Designs gerät. Wie Bild 4.9 zeigt, wird in dem Beispiel die Abfrage des AD-Wandlers sowie der Vergleich mit den Grenzen in dem PLD-Teil implementiert, der damit das komplette Interface zum ADC enthält. Der Mikrocontroller wird lediglich dann unterbrochen, wenn die Grenzwertverletzung auftritt und somit eine 'echte' Behandlung notwendig ist.

Zur Unterbringung von Ereignisbehandlungen ist natürlich auch hergestellte Hardware (ASIC) geeignet, dies stellt lediglich eine Frage der Herstellungszahlen und –kosten dar. Für den Jitter und die Bearbeitungszeiten der Hardware-Routinen kann man allgemein sagen, dass diese in der Größenordnung eines oder weniger Takte liegt.

4.5 Zusammenfassung der Zeitkriterien für lokale Systeme

Aus den bisherigen Betrachtungen lässt sich resümieren, dass einige Zeitkriterien existieren, die die Behandlung und die Implementierungsart entscheidend beeinflussen. Im Wesentlichen sind dies drei Kriterien, die aus der Prozessumgebung stammen:

- Der zeitliche Jitter *Tjt* (auch als maximale Latenzzeit zu bezeichnen, siehe Definition 2.6 und 3.4) gibt diejenige Zeit an, mit der der Start der Reaktionsroutine schwanken darf. Gründe hierfür können zeitsynchrone Aktivitäten sein, für die nur geringe Abweichungen akzeptierbar sind. Liegt dieser Jitter unterhalb ca. 10 Befehlsausführungszeiten, so kann mit Sicherheit davon ausgegangen werden, in einem für Prozessoren kritischen Bereich zu liegen.

  Die unkritische Grenze, ab der also mit einem garantierten Verhalten des Prozessors zu rechnen ist, ist natürlich individuell von dem System abhängig. In jedem Fall ist das System sicher konzipiert, wenn der erlaubte Jitter größer ist als die Summe aller höherpriorisierten Ereignisse (unter Einbezug der Auftrittshäufigkeit) bei Ereignis-gesteuerten Systemen bzw. die Zykluszeit bei Zeitgesteuerten Systemen.
- Die Servicezeit $T_{Service}$ spielt eine scheinbar unwichtige Rolle, da sie ja sowieso eingeplant werden muss. Bei Servicezeiten, die mehr als 30% der gesamten Rechenzeit (im Normalfall oder Worst Case) einnehmen, muss man jedoch davon ausgehen, dass diese Zeit sehr dominant ist und die übrigen Teile des Systems stark beeinflusst. Diese 30%-Grenze ist allerdings unscharf, während Servicezeiten < 1% sicher keinen Einfluss nehmen.

	Kritischer Wert	Unkritischer Wert
T_{Jitter}	< 10 Befehlszeiten	> $\sum$ alle höherpriorisierten Reaktionszeiten (Ereignis-gesteuert) oder > Zykluszeit (Zeit-gesteuert)
$T_{Service}$	> 50 % der gesamten Rechenzeit	< 1 % der gesamten Rechenzeit
$T_{Reaction}$	< 100 Befehlszeiten	> $\sum$ alle höherpriorisierten Reaktionszeiten (Ereignis-gesteuert) oder > Zykluszeit (Zeit-gesteuert)

Tabelle 4.1 Zusammenfassung der charakteristischen Zeiten von Ereignissen

- Die maximal geforderte Reaktionszeit $T_{Reaction}$ setzt sich aus der Latenzzeit und der Servicezeit zusammen, allerdings müssen noch mögliche Unterbrechungen

mitbetrachtet werden. Kritisch wird es für die Reaktionszeit, wenn diese etwa < 100 Befehlsausführungszeiten ist (die Grenze ist auch hier wieder individuell). Die unkritische Grenze wird wieder bei der Summe über alle Reaktionszeiten bzw. der Zykluszeit erreicht.

Tabelle 4.1 fasst die charakteristischen Zeiten zusammen. Ist für eine von diesen Zeiten für ein konkretes System die kritische Grenze erreicht, so ist der Systemdesigner aufgefordert, exklusive Hardware hierfür bereitzustellen. Sind hingegen alle Zeiten unkritisch, kann man zu einem Sharing-Betrieb übergehen. Im Zwischenbereich hingegen muss individuell konzipiert werden, was die geeignete Wahl darstellt (siehe Bild 4.10).

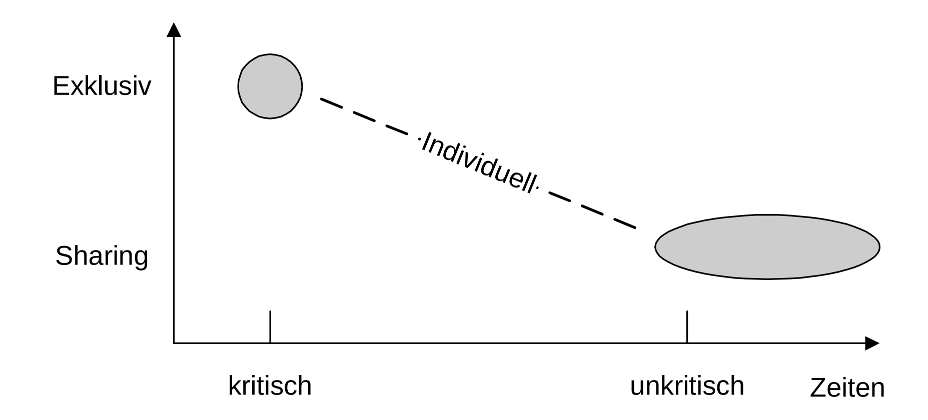


Bild 4.10 Designraum exklusiv/sharing für Systeme mit zeitlichen Randbedingungen

4.5.1 Vergleich Zeit-Steuerung und modifizierte Ereignis-Steuerung

Aus den Überlegungen zu den charakteristischen Zeiten kann man weiterhin eine Unterscheidung für Zeit-gesteuerte und modifizierte Ereignis-gesteuerte Systeme ableiten. Beide sind in der Lage, deterministische Echtzeitbedingungen zu erfüllen.

Der erste Vorteil für die modifizierten Ereignis-gesteuerten Systeme besteht darin, dass die Abfrage der Inputleitungen, die bei der Zeit-Steuerung im Pollingverfahren erfolgen muss, durch die Ereignissignale entfällt. Selbst wenn der Kontextwechsel in eine Interrupt Service Routine einen vergleichbaren Aufwand zum Polling darstellt, ist der Gewinn an Rechenzeit genau dann vorhanden, wenn die Ereignisse nicht regelmäßig kommen. Wie an einem kleinen Modell gezeigt werden wird, ist dieser Gewinn vergleichsweise gering.

Für den maximal zulässigen Jitter gilt, dass dieser nicht überschritten werden darf. Dies bedeutet, dass im Zeit-gesteuerten System entweder die Zykluszeit hiernach

auszurichten ist, oder für die kritischen Teile muss eine Behandlung mehrfach innerhalb eines Zeitzyklus durchlaufen werden. Lösung 1 führt dazu, das System für alle Teile mit einer Zykluszeit zu versehen, die nicht überall benötigt wird, Lösung 2 erhöht die Pollingrate nur individuell, bedeutet aber auch mehr Programmaufwand intern.

Die Kombination der Einsparung von Rechenzeiten bei ereignislosen Abschnitten und der individuellen Anpassung der Latenzzeiten ergibt die wesentlichen Vorteile für modifizierte Ereignis-gesteuerte Systeme. Um dies zu quantifizieren, sei hier ein Modell angegeben:

- Es wird ein RISC-basiertes Mikroprozessorsystem gewählt, bei dem – vereinfachend – 1 Befehl/Takt als Geschwindigkeit angenommen wird.
- Jeder IRQ wird durch einen IRQ-Controller priorisiert, und der Sprung in die Interrupt Service Routine ISR wird mit 10 Takten veranschlagt (in der Reaktionszeit enthalten). Jeder höherpriorisierte IRQ (0: höchste Priorität) kann niedrigere ISR unterbrechen, das Hauptprogramm ist jederzeit unterbrechbar.
- Es werden 4 Zustände des Prozesses angenommen: Priorität 0 benötigt 100 Takte zur Bearbeitung, Priorität 1 200, Priorität 2 300 und Priorität 3 400 Takte (alles Maximalwerte). Für das Zeit-gesteuerte System sollen diese Zustände nacheinander abgefragt und bearbeitet werden, wobei die Abfrage 10 Takte beanspruchen soll.
- Für die Häufigkeit der IRQs wird angenommen, dass sie alle mit maximaler Frequenz von 10 kHz auftreten können. Zur Simulation wird eine Variation angenommen, also beim ersten Mal nach 100 µs, beim zweiten Mal nach 200 µs, dann nach 300 µs, wieder nach 10 µs usw.
- Als maximal zulässige Reaktionszeiten werden für Priorität 0 20 µs, für 1 50 µs und für die anderen 100 µs angenommen.
- Für das Verhältnis von ISR und Hauptroutinenrechenzeit wird ca. 1:1 angenommen, d.h., der Zyklus soll auf 2000 Takte ausgelegt werden. Für das Hauptprogramm wird gefordert, dass im zeitlichen Mittel ca. $2 * 10^6$ Instruktionen pro Sekunde ausführbar sind.

Für dieses Modell ergeben sich dann folgende, in Tabelle 4.2 wiedergegebene Schätzungen. Die darin vorhandenen Ergebnisse können so zusammengefasst werden:

- Das Verhältnis der Taktanzahlen, die dem Hauptprogramm zur Verfügung stehen, ist nahezu konstant, d.h., der Overhead für die Zeitsteuerung ist vergleichsweise gering. Es verbessert sich zwar noch weiterhin, wenn die IRQs noch sporadischer auftreten, dennoch dürfte der Effekt auf wenige Prozent begrenzt bleiben.
- Der Unterschied in der Auslegung der Betriebsfrequenz ist sehr groß. Die Ursache hierfür liegt in der geforderten maximalen Reaktionszeit. Die notwendige

Frequenz kann für die Zeitsteuerung dadurch heruntergesetzt werden, dass die extrem kritischen Teile mehrfach vorkommen oder die Zykluszeit verringert wird. Letztere Maßnahme ist begrenzt (die Berechnungszeit der Prozess-gekoppelten Software setzt das Limit), ein häufigerer Timer-IRQ erzeugt mehr Overhead (Klammerwerte in Tabelle 4.2: Für Priorität 0 werden zwei Timer-IRQ pro 2000 Takte Zyklus erzeugt).

	Time-triggered	Modified Event-triggered
Anzahl Takte für Prozess-gekoppelte Software (auf 12000 Takte)	3300 (3420)	3120
Anzahl Takte Hauptprogramm (auf 12000 Takte)	8700 (8580)	8880
Relativer Gewinn	1	1.02 (1.035)
Maximale Latenzzeit [Takte]	0: 2000 (1000) 1: 2100 2: 2300 3: 2600	0: 0 1: 100 2: 300 3: 600
Mittlere Latenzzeit [Takte]	0: 1000 (500) 1: 1050 2: 1150 3: 1300	0: 0 1: 2,5 2: 22,6 3: 90
Maximale Reaktionszeit [Takte]	0: 2100 (1100) 1: 2300 2: 2600 3: 3000	0: 100 1: 300 2: 600 3: 1000
Resultierende Taktfrequenz [MHz]	105 (55) MHz (Reaktionszeit Priorität 0)	10 MHz (Reaktionszeit Priorität 3)

Tabelle 4.2 Taktzahlen und Operationsfrequenz im Modellsystem (Zahlen in Klammern: Erweitertes Timer-IRQ-System für Priorität 0 mit zwei Serviceroutinen pro Zyklus)

Als Fazit dieses Vergleichs bleibt an dieser Stelle festzuhalten, dass die (modifizierten) Ereignis-gesteuerten Systeme insbesondere Forderungen nach kurzen Reaktionszeiten wesentlich besser erfüllen können. Die Dimensionierung des Zeitgesteuerten Systems ist in dem Modell gerade deshalb so hoch, weil die Reaktionszeit der höchsten Priorität zwar weit von der für die Befehlsbearbeitung entfernt ist, jedoch die Zykluszeit dieser Größe angepasst werden muss.

Eine Schätzung des Effekts durch Einführung von 'Modified Event-triggered with Exception Handling' kann für das Modell ebenfalls gegeben werden. Verringert man die Arbeitsfrequenz beispielsweise auf 8 MHz, so kann für aller Prioritäten die Echtzeitbedingung eingehalten werden, lediglich für Priorität 3 ist dies nicht immer möglich. Hier wird nun im Ausnahmefall (drohende Zeitüberschreitung) eine Notroutine angesprungen, die eine vorläufige Reaktion darstellt.

Das Down-Scaling in diesem Fall führt zu Einsparungen von ca. 20%. Dies ist im Einzelfall zu überprüfen und stellt lediglich eine erste Schätzung dar.

4.5.2 Übertragung der Ergebnisse auf verteilte Systeme

Das Wesen der verteilten Systeme – die Einbindung und der Zugriff auf ein nicht-exklusives Kommunikationsmedium – erfordert eine gesonderte Behandlung, bedingt eben durch die Nicht-Exklusivität. Ein derartiges System kann so ausgelegt sein, dass der jeweils lokale Teil auf Basis einer modifizierten Ereignissteuerung läuft, die Kommunikation ggf. jedoch entkoppelt davon.

Auf Seiten des Netzwerks muss ein deterministisches Verfahren zur Buszuteilung existieren, das zumindest für einen Satz von Nachrichten die echtzeitfähige Übertragung garantiert. Hier folgt eine kurze Diskussion der Zuteilungsverfahren:

- CSMA/CD (Carrier Sense Media Access with Collision Detection): Dieses bei Ethernet verwendete Verfahren scheidet aus, da der Zugriff probabilistisch ist und somit keine maximale Übertragungszeit garantiert werden kann.
- CSMA/CA (Carrier Sense Media Access with Collision Avoidance): Das Controller-Area Network (CAN) verwendet dieses Verfahren, bei dem bei einem Zugriff eine Kollision vermieden wird. Dies bedeutet, dass ohne weitere Maßnahmen die höchste Priorität garantiert übertragen wird, alle anderen aber wiederum keine Echtzeitfähigkeit besitzen.

  Die besonderen Maßnahmen können die maximale Wiederholungsfrequenz betreffen. Durch diese Einschränkung könnte ein CSMA/CA-Netzwerk echtzeitfähig werden. Dadurch wäre ein Ereignis-gesteuertes Netzwerk tatsächlich möglich!
- TTP/C (Time-Triggered Protocol Class C): In diesem Zeit-gesteuerten Protokoll besitzen alle Knoten eine gemeinsame Zeit mit geringem Jitter. Dies wird durch spezielle Verteilung erreicht. Über eine Zeittabellen-gesteuerte Nachrichtensendung erhält jeder Knoten eine garantierte Sendemöglichkeit, außerdem können alle anderen Knoten die Betriebsfähigkeit des sendenden erkennen (und vor allem auch den Ausfall!).
- Byte Flight: Das Byte Flight Protokoll benötigt einen ausgezeichneten Sender, der über ein Zeitsignal eine gemeinsame Zeit verteilt. Diese gemeinsame Zeitbasis (Jitter: 100 ns) veranlasst die anderen Knoten nacheinander, Pakete zu senden oder ruhig zu bleiben. Dadurch wird es möglich, für eine begrenzte Anzahl von Sendungen einen exklusiven Zugriff zu gestatten.

  Der Rest in einem Zeitschlitz wird nach dem CSMA/CA-Verfahren verteilt, sodass der Bus optimal ausgenutzt wird und zugleich (für eine begrenzte Anzahl von Daten) echtzeitfähig ist.

4.5.3 Verteilung der Zeit in verteilten Systemen

Letztendlich steht und fällt die Echtzeitfähigkeit in Time-Triggered-Kommunikationssystemen mit der Verteilung einer gemeinsamen Zeit. Hier wurde bei IEEE ein präzises Zeitprotokoll definiert (Precision Time Protocol, IEEE-1588, [GM03] [IEE1588]), mit dessen Hilfe diese Verteilung erfolgen kann.

Die Verteilung erfolgt so, dass eine Clock in dem zu betrachtenden Netzwerk als Master bezeichnet wird. Diese Uhr soll möglichst genau sein, ggf. Anschluss an exakte Zeitgeber haben usw. Der Master sendet nun eine spezielle Meldung als Broadcast aus, die *Sync Message*. Diese Meldung enthält einen Zeitstempel, insbesondere eine Schätzung, wann sie auf dem Netzwerk sein wird.

Falls hohe Präzision gefordert (und möglich) ist, wird die Sync Message von einer zweiten Meldung, der *Follow-Up Message*. Diese enthält dann die tatsächlich gemessene Zeit der Übertragung, also des physikalischen Zugriffs auf das Medium Netzwerk. Misst nun der Slave die Empfangszeit mit entsprechender Präzision, kann er die interne Uhr auf den Master abstimmen – mit der Ausnahme, dass die Übertragungszeit nicht berücksichtigt wurde.

Diese Übertragungszeit kann ebenfalls bestimmt werden. Die Slaves, die diese Sendung empfangen haben, müssen nun mit allerdings geringerer Häufigkeit diese Prozedur wiederholen, indem sie wieder eine Sync Message und ggf. eine Follow-Up Message senden, nun nur an den Master adressiert. Hierin wird die Übertragungszeit der Master-Slave-Abstimmung ebenfalls übermittelt, und nun stehen beide Messungen, hin- und Rückweg, zur Verfügung.

Unter der Annahme, dass die Übertragung eine symmetrische Latenzzeit aufweist, kann nun also auch diese Zeit bestimmt werden. Die Synchronisation reicht hierdurch bis in den Sub-Mikrosekundenbereich zurück, allerdings müssen Router aufgrund ihrer langen Verzögerung ausgeschlossen werden (hierzu bietet IEEE-1588 ebenfalls Methoden an).

5 Eingebettete Systeme und Verlustleistung

Dieses Kapitel dient dem Zweck, den Zusammenhang zwischen den Systemen, die programmiert werden können, den Entwurfssprachen und den in Kapitel 1 bereits diskutierten Randbedingungen darzustellen.

Hierzu wird der quantitative Zusammenhang zwischen Fläche *A*, Zeit *T* und Verlustleistung *P* untersucht. Dieser Zusammenhang dürfte existieren, die Quantifizierung ist interessant. Hat man nun mehrere Möglichkeiten, kann man das Design optimieren. Man spricht dann auch von dem *Designraum*.

5.1 Der quantitative Zusammenhang zwischen Rechenzeit, Siliziumfläche und Verlustleistung

Rechenzeit und Siliziumfläche

Folgende Gedankenkette zeigt einen zumindest qualitativen Zusammenhang zwischen Zeit und Fläche. Für einen 8-Bit-Addierer existieren viele Implementierungsmöglichkeiten:

- Sequenziell: 1-Bit-Addierer mit Shift-Register als Speicher, getaktete Version. Dieser Addierer berechnet in einem Takt nur ein Summenbit sowie das Carry-Bit, beide werden gespeichert und weiter verwendet.
- Seriell: Ripple-Carry-Adder, 8*1-Bit-Addierer mit seriellem Übertrag. Dieser Addierer ist die bekannte Form und wird gelegentlich auch als sequenziell bezeichnet.
- Total parallel: Addierschaltung, bei der alle Überträge eingerechnet sind. Hier ist die Berechnungszeit unabhängig von der Breite der Eingangswörter.
- Carry Look-Ahead Adder: Zwei Schaltnetze, eines für Carry, ein folgendes für die Addition. Hier wird zwar die im Vergleich zum total parellelen Addierer doppelte Zeit benötigt, aber immer noch unabhängig von der Datenbreite.
- Zwischenformen wie 4*2-Bit-Paralleladdierer usw.

Bild 5.1 zeigt reale Werte für einen 12-Bit-Addierer. Als Standardverzögerungszeit sind 10 ns pro Gatter angenommen, zur Flächenbestimmung wurde die Zahl der Terme (Disjunktive Normalform DNF) herangezogen.

Hieraus und aus anderen Schaltungen kann man zunächst empirisch schließen, dass es für begrenzte Schaltungen ein Gesetz wie

$$A \cdot T^k = \text{const(technology)} \;\; \text{mit} \;\; k = 1..2 \tag{5.1}$$

gibt. Dieses Gesetz ist zwischenzeitlich auch theoretisch bestätigt worden. Die Exponenten k tendieren für arithmetische Operationen gegen 2.

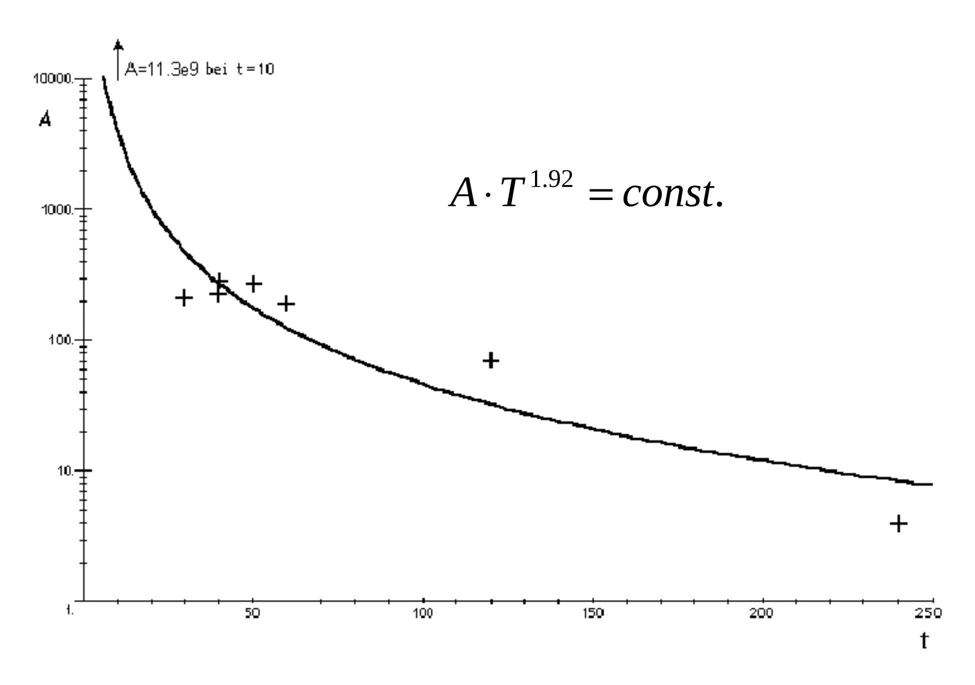


Bild 5.1 AT-Gesetz für 12-Bit-Addierer, verschiedene Implementierungsvarianten

Interpretation: Es liegt hier eine Trade-Off-Funktion vor, die verdeutlichen soll, dass man – je nach Randbedingungen – ein applikationsspezifisches Optimum finden kann.

Weiterhin können einzelne Implementierungen von diesem Zusammenhang signifikant abweichen. Man kann daher die durch diese Funktion gezogene Grenze als Optimalitätskriterium heranziehen, so dass Punkte unterhalb der Kurve (siehe auch Bild 5.1) optimal sind.

Definition 5.1:

Die Flächen-Zeit-Effizienz (space-time-efficiency) $E_{S/T}$ ist definiert als

$$E_{S/T} = \sqrt{\frac{1}{A \cdot T^2}} = \frac{1}{\sqrt{A} \cdot T}$$

Während das A*T^k-Gesetz als Zusammenhang für eng begrenzte Operationen, also etwa einen Addierer gefunden wurde, wird es aktuell auch zur Beurteilung ganzer ICs benutzt, beispielsweise für Mikroprozessoren.

Rechenzeit und Verlustleistung

Der Zusammenhang zwischen Verlustleistung und Rechengeschwindigkeit kann etwas genauer betrachtet (und auch hergeleitet) werden. Bei einem CMOS-Design, wie es für Mikroprozessoren State-of-the-Art ist, zählen 3 Komponenten zur Verlustleistung hinzu:

$$P_{total} = P_{SC} + P_{leakage} + P_{switching_losses} \tag{5.2}$$

P_{SC} (Short Current, Kurzschlussstrom) resultiert aus demjenigen Strom, der kurzzeitig beim gleichzeitigen Umschalten beider Transistoren eines CMOS-Paares fließt. Dies ist prinzipbedingt im CMOS-Design verankert, und die Anzahl der Umschaltungen pro Zeiteinheit ist natürlich proportional zum Takt.

$$P_{SC} = V \cdot I_{SC} \tag{5.3}$$

$P_{leakage}$ (Leakage Current, Leckstrom) entstammt aus dem dauerhaft fließenden Leckstrom einer elektronischen Schaltung. Dieser Strom ist bei CMOS-Schaltungen natürlich sehr klein, weil in jedem Stromkreis mindestens ein Transistor sperrt, er ist aber nicht 0. Aufgrund der enormen Anzahl an Transistoren in aktuellen Schaltungen sowie der ständigen Verkleinerung der Strukturen summieren sich die Ströme zu mittlerweile signifikanten statischen Verlustleistungen:

$$P_{leakage} = V \cdot I_{leakage} \tag{5.4}$$

$P_{switching_losses}$ (Switching Losses, Schaltverluste) ist derjenige Anteil, der aktuell als dominant betrachtet wird. Dieser Anteil entstammt dem Umladestrom, der durch das Laden und Entladen der Transistorkapazitäten entsteht. Die daraus resultierende *mittlere* Verlustleistung ist bei gegebener Umladefrequenz f

$$P_{switching_losses} = \frac{C}{2} \cdot V^2 \cdot f \tag{5.5}$$

Vernachlässigt man insbesondere den statischen Verlustleistungsanteil – ein Vorgang, den man bei einigen höchstintegrierten Schaltungen bereits nicht mehr machen kann –, dann gilt der bekannte Zusammenhang, dass bei konstanter Spannung die Verlustleistung P linear mit der Frequenz f steigt.

Also ein linearer Zusammenhang zwischen Verlustleistung und Rechengeschwindigkeit? Nein, denn Gl. (5.5) gilt bei konstanter Spannung, und genau diese Betriebsspannung lässt sich bei sinkender Betriebsfrequenz in modernen CMOS-Schaltungen ebenfalls absenken. Um diesen Effekt zu quantifizieren, sei folgende Ableitung gegeben:

Die Kapazität C im Transistor bleibt konstant und muss beim Umschalten geladen werden. Die dafür notwendige Ladungsmenge ist

$$Q = C \cdot V = I \cdot t_{\min} = \frac{I}{f_{\max}} \tag{5.6}$$

Der Ladestrom I ist von der Betriebsspannung und der Schwellenspannung V_{th} (Threshold-Voltage) abhängig. Diese Abhängigkeit ist etwas komplexer, aktuell wird folgende Näherung angenommen:

$$I = const. \cdot (V - V_{th})^{1,25} \tag{5.7}$$

Die maximal mögliche Frequenz ergibt sich durch Einsetzen von (5.7) in (5.6) und Auflösung nach f_{max}. Hierbei kann eine weitere Näherung für den Fall angenommen werden, dass V von V_{th} weit genug entfernt ist:

$$f_{\max} = const_1 \cdot \frac{(V - V_{th})^{1,25}}{V} \approx const_2 \cdot V \quad (\mathit{für}\,(V - V_{th}) \geq V_{th}) \tag{5.8}$$

Diese Formel sagt also aus, dass mit der Skalierung der Betriebsspannung V auch die maximale Betriebsfrequenz f_{max} skaliert. Insgesamt gilt mit allen Näherungen der quantitative Zusammenhang

$$P \cdot T^3 = const. \tag{5.9}$$

Interpretation: Dieser Zusammenhang zeigt auf, wie Verlustleistung und Rechengeschwindigkeit sich gegenseitig beeinflussen, wenn Betriebsspannung und Frequenz verändert werden dürfen. Der gewaltige Zuwachs der Verlustleistung (bei verdoppelter Frequenz 8fache Verlustleistung) ist sehr signifikant.

Bild 5.2 zeigt den Zusammenhang zwischen P, A und T (in qualitativer Form). Es wird für die Zukunft angenommen, dass Server-Architekturen optimiert auf Rechengeschwindigkeit, Architekturen für eingebettete Systeme jedoch mehr auf Verlustleistungsminimierung (und damit Flächenminimierung) ausgelegt sein werden.

Anmerkung: Die Reduzierung der Strukturbreiten in den ICs haben aktuell Auswirkungen auf die Betriebsspannung und die Verlustleistung. Durch die kleiner werdenden Strukturen muss die Betriebsspannung gesenkt werden. Dies führt auch zu sinkenden Thresholdspannungen, was wiederum zu drastisch steigenden statischen Verlustleistungen führt. Die Herleitung, insbesondere der Teil nachdem (5.5) den einzigen nennenswerten Beitrag zur Verlustleistung liefert, gilt dann zukünftig nicht mehr. Es kann sogar so sein, dass die statische Verlustleistung überwiegt.

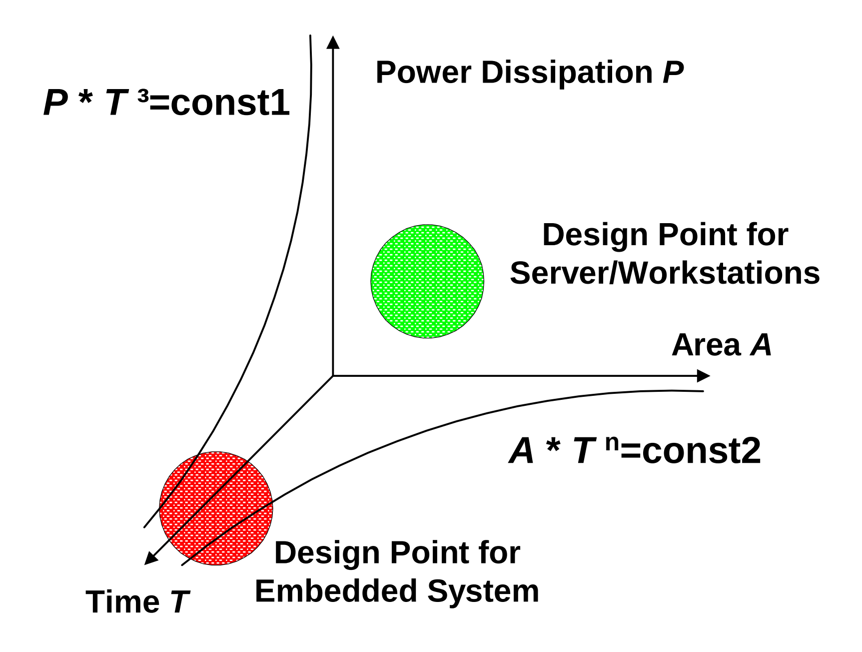


Bild 5.2 Zusammenhang zwischen *P*, *T* und *A*

5.2 Ansätze zur Minderung der Verlustleistung

Wie bereits in Abschnitt 5.1 gezeigt wurde, existiert ein quantitativer Zusammenhang zwischen Verlustleistung und Rechenzeit. Das dort abgeleitete Gesetz, dass $P*T^3$ = const. gelten soll, gilt allerdings nur unter der Voraussetzung, dass man sich in einem Design (sprich: eine Architektur) bewegt und Versorgungsspannung sowie Taktfrequenz ändert.

Das ist natürlich auch eine Methode, aber eben nur eine, die zur Verlustleistungsminderung in Frage kommt. In der Realität sind es 4 Methoden, die zur Anwendung kommen:

- Auswahl einer Architektur mit besonders guten energetischen Daten
- Codierung von Programmen in besonders energiesparender Form
- Einrichtung von Warte- und Stoppzuständen
- Optimierung der Betriebsfrequenz und Betriebsspannung nach Energiegesichtspunkten

Und um es vorweg zu nehmen: Dies ist ein hochaktuelles Forschungsgebiet, es gibt Ansätze [BBM00], aber noch keinerlei analytische Lösungen. Im Folgenden sollen diese Ansätze kurz diskutiert werden.

5.2.1 Auswahl einer Architektur mit besonders guten energetischen Daten

Es mag auf den ersten Blick natürlich unwahrscheinlich erscheinen, warum einige Architekturen mehr, andere weniger Verlustleistung (bei gleicher Geschwindigkeit) benötigen, dennoch stellt sich in der Praxis immer wieder heraus, dass es drastische Unterschiede bei Mikroprozessoren und Mikrocontrollern gibt [Bro+00].

Bild 5.3 zeigt einige Mikroprozessoren im Vergleich [Bro+00]. Hierzu wurden die erhältlichen SpecInt2000-Werte pro eingesetzter elektrischer Leistung – bezogen auf den ältesten (und schlechtesten) Sparc-III-Prozessor– dargestellt, und zwar als $(SPEC)^x/W$ mit $x = 1 \ldots 3$. Die unterschiedliche Metrik war bereits in den Darstellungen aus Abschnitt 5.1 sichtbar: Ist nun P*T konstant oder $P*T^3$?

Diese Unterschiede sind in der unterschiedlichen Mikroarchitektur begründet, manchmal auch darin, dass viel Kompatibilität mitgeschleppt wird. Bild 5.7 zeigt allerdings nur die Hälfte der Wahrheit, indem kommerzielle Mikroprozessorprodukte miteinander verglichen werden.

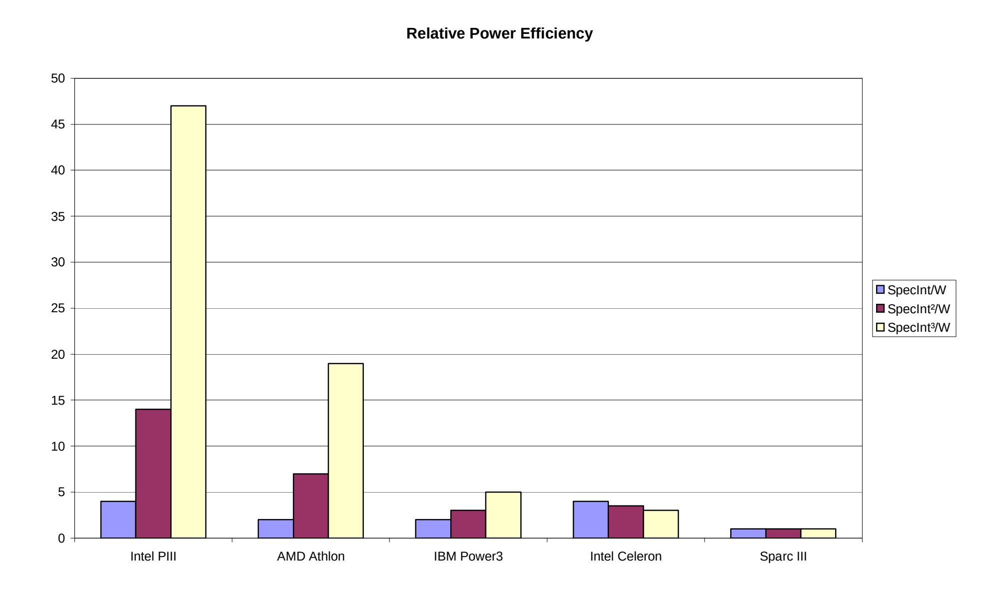


Bild 5.3 Relative Leistungseffizienz im Vergleich

In [MP01] werden zwei Produkte etwa gleichen Erscheinungsdatums miteinander verglichen: Ein AMD Mobile K6 und ein Intel Xscale-Mikrocontroller, der von der ARM (Advanced RISC Machine) StrongARM-Architektur abgeleitet wurde. Der AMD Mobile K6 benötigt bei 400 MHz eine elektrische Leistung von 12 W, der Xscale bei 600 MHz nur 450 mW! Nimmt man grob an, dass beide etwa gleich

schnell arbeiten (aufgrund der Superskalarität im AMD-Prozessor ist dieser bei gleicher Arbeitsfrequenz schneller), ergibt dies ein Verhältnis der elektrischen Leistung von ca. 1:27!

Welches Fazit kann man hieraus ziehen? Die aktuelle Entwicklung der integrierten Schaltkreise geht mehr in die Richtung Leistungseffizienz, nicht mehr Performance. Dies wurde bereits in Bild 5.2 angedeutet, und derzeit sind große Bemühungen zu verzeichnen, diese Effizienz noch zu steigern.

Dies betrifft das Hardwaredesign, und der Systemdesigner kann als Anwender nur die geeignete Architektur auswählen. Ist die Leistungsbilanz bei einem Design im Vordergrund stehend oder auch nur eine wesentliche Randbedingung, sollte man mit der Auswahl des Mikroprozessors/Mikrocontrollers anhand der Daten beginnen und alle anderen Werte wie Betriebsfrequenz usw. als nachrangig betrachten.

5.2.2 Codierung von Programmen in besonders energiesparender Form

Vor einigen Jahren war ein Thema wie energiesparende Software undenkbar, mittlerweile hat es sich jedoch schon etabliert [SWM01]: Man kann die spezifische Leistungsaufnahme pro Befehl bestimmen und dann auswählen, welcher tatsächlich ausgeführt werden soll – falls es Variationsmöglichkeiten gibt. Kandidaten hierfür sind z.B. Multiplikationsbefehle und deren Übersetzung in eine Reihe von Additionsbefehlen.

Insbesondere die Multiplikation einer Variablen mit einer Konstanten kann in diesem Beispiel als möglicher Kandidat gelten. Die Multiplikation mit 5 z.B. wird dann auf einen zweifachen Shift nach links (= Multiplikation mit 4) und anschließender Addition mit dem ursprünglichen Wert ausgeführt, wenn dies energetisch günstiger sein sollte (siehe Bild 5.4).

```
mov   R3, #5         ;                  asl   R3, R5         ; * 2
mul   R3, R3, R5     ; 5 * (R5)  --->   asl   R3, R3         ; * 4
                                        add   R3, R3, R5     ; 5 * (R5)
```

Bild 5.4 Umsetzung einer Multiplikation mit Konstanten in energetisch günstigere Form

Um dies wirklich auszunutzen, muss die Hilfe eines Compilers in Anspruch genommen werden. Derartige Ansätze sind in der Forschung vertreten, z.B. dargestellt in [SWM01]. Es dürfen jedoch keine Größenordnungen an Energieeinsparung dadurch vermutet werden, die Effekte bleiben im Rahmen einiger 10%.

5.2.3 Einrichtung von Warte- und Stoppzuständen oder Optimierung der Betriebsfrequenz?

Eine andere Möglichkeit zur Energieeinsparung entsteht durch die Einführung von verschiedenen Betriebsmodi insbesondere von Mikrocontrollern. Diese Modi, im Folgenden mit RUN, IDLE und SLEEP bezeichnet, bieten neben variiertem Funktions- und Reaktionsumfang auch differierende Energiebilanzen. Bild 5.5 zeigt ein Beispiel aus [BBM00] für den Intel StrongARM SA-1100 Mikroprozessor.

Fehler! Keine gültige Verknüpfung.

Bild 5.5 Power State Machine für SA-1100

Der Übergang von RUN in IDLE sowie RUN in SLEEP erfolgt üblicherweise durch Software. Hier können spezielle Instruktionen oder das Setzen von Flags zum Einsatz kommen. Im IDLE-Modus ist die Taktversorgung prinzipiell eingeschaltet, insbesondere eine vorhandene PLL, und die Peripherie eines Mikrocontrollers bleibt meist ebenfalls versorgt. Aus diesem Grund können Ereignisse im IRQ-Controller wahrgenommen werden und führen zum Aufwecken des Prozessorkerns.

Im SLEEP-Modus wird die Taktversorgung komplett ausgeschaltet, die PLL ist ausgeschaltet. Dadurch sinkt die Leistungsaufnahme nochmals, auch die peripheren Elemente werden ausgeschaltet. Der Nachteil ist derjenige, dass das Starten des Prozessors/Controllers jetzt recht lange dauert, weil die PLL sich erst wieder einphasen muss. Außerdem können nur noch asynchrone Ereignisse wahrgenommen werden, meist ist dies ein singuläres Ereignis, z.B. der Non-Maskable Interrupt (NMI) oder der Reset.

Die eigentliche Schwierigkeit mit der Power-State-Machine besteht darin, Kriterien zu finden, wann in welchen Zustand übergegangen werden kann. Man denke dabei nur an die verschiedenen Energiesparmodi bekannter Rechner. In Bild 5.5 ist es so, dass der Übergang nur Zeit, keine Leistung kostet. Dies kann im allgemeinen Fall jedoch anders sein, und ein verkehrtes Abschalten könnte sogar zu erhöhter Verlustleistung führen.

Zurzeit sucht man nach neuen Methoden, die die Übergänge definieren. Für den Systementwickler stellt dies natürlich eine gute Methode dar, unter der allerdings die Echtzeitfähigkeit leiden dürfte. Meist sind jedoch Echtzeitsysteme nicht unbedingt batteriebetrieben, energiesparend sollten sie jedoch trotzdem sein.

Die andere Methode wäre diejenige, auf die Power-State-Machine zu verzichten und die Betriebsfrequenz an das untere Limit zu fahren. Den Netteffekt erfährt man für einen Vergleich nur durch intensive Simulationen, und auch hier dürfte die Echtzeitfähigkeit ggf. leiden.

5.2.4 Neue Ansätze zur Mikroprozessor-Architektur: Clock-Domains und GALS-Architektur

Eine optimale Lösung in Richtung minimaler Energieumsatz bei der Programmausführung wäre es, wenn Betriebsspannung und –frequenz den aktuellen Anforderungen angepasst werden können. In [TM05] wird ein derartiger Ansatz diskutiert, und zwar in einer vergleichsweise feinkörnigen Form.

Die Idee zielt eigentlich auf das Design superskalarer Prozessoren [Sie04]. Diese Prozessoren, die in der Regel sehr groß und damit auch auf der Siliziumfläche ausgedehnt sind, haben besondere Probleme mit einer gleichmäßigen Taktverteilung (ohne Skew), die entweder sehr viel Verlustleistung oder eine Verlangsamung mit sich bringt. Der in [TM05] vorgestellte Ansatz zeigt nun, dass synchrone Inseln, asynchron untereinander verbunden, die bzw. eine Lösung hierfür darstellen.

Diese Architektur wird GALS, Globally Asynchronous Locally Synchronous, genannt. Die lokalen Inseln werden jeweils mit einem Takt (Clock Domain) versorgt, der nun sehr genau an den aktuellen Rechenbedarf angepasst werden kann (Hardware: VCO, Voltage Controlled Oscillator mit DVS, Dynamic Voltage Scaling). Wie aber kann man sich die asynchrone Kommunikation vorstellen?

Asynchron ist eigentlich das falsche Wort hierfür, selbst-synchronisierend ist richtig. Hiermit ist gemeint, dass über die Kommunikationsleitungen nicht nur Daten (und ggf. ein Takt) geführt werden, sondern dass mit den Daten ein Handshake verbunden ist. In etwa verläuft dies nach dem Handshake:

1. (S:) Daten sind gültig
2. (E:) Daten sind übernommen
3. (S:) Daten sind nicht mehr gültig
4. (E:) Wieder frei für neue Daten

Hiermit ist grundsätzlich ein Verfahren möglich, wie die Ausführung von Programmen (Energie- bzw. Verlustleistungs-) optimal angepasst werden kann.

Abschnitt II: Software Engineering für Eingebettete Systeme

6 Einführung in die Sprache C

Die nachfolgende Einführung in die Sprache C ist in wesentlichen Teilen [CKURS] entnommen. C stellt eine sehr populäre, imperative Sprache dar, die sich durch folgende Eigenschaften auszeichnet:

- relativ kleiner Sprachkern, kompakte Notation
- reichhaltiger Satz von Standarddatentypen
- reichhaltiger Satz von Operatoren
- Zeiger, Felder, Verbünde für komplexe Datenstrukturen
- gute Abbildung dieser auf Maschinenebene: hohe Effizienz
- alles andere wie E/A, Speicherverwaltung etc. ist in Standard-Bibliothek untergebracht
- wegen Einfachheit und Verbreitung extrem hohe Portabilität

Die immense Flexibilität und Ausdrucksstärke von C birgt aber auch größte Gefahren in der Hand eines unerfahrenen oder leichtfertigen Programmierers: C ist definitiv keine sichere Sprache, daher ist größte Disziplin geboten, um Fehler zu vermeiden. Der Leitsatz von Ritchie beim Entwurf der Sprache lautete: ”Trust the programmer!“.

Allerdings muss man auch sagen, dass bei entsprechender Programmierdisziplin C auch für sichere Software geeignet ist. Im Anschluss an die Einführung ist daher ein Abschnitt zu Codierungsregeln beigefügt, nähere Literatur siehe [Hat95].

Zunächst jedoch zum Begriff der imperativen Sprache. **Imperative Programmierung** ist ein Programmierparadigma. Ein imperatives Programm beschreibt eine Berechnung durch eine Folge von Anweisungen, die den Status des Programms verändern. Im Gegensatz dazu wird in einer deklarativen Sprache bzw.Programm eine Berechnung beschrieben, in der codiert wird, *was* berechnet werden soll, aber nicht *wie*.

Mit anderen Worten: In imperativen Sprachen wie *C* werden die Algorithmen bis ins letzte Detail so beschrieben, wie sie auch auszuführen sind.

Im weiteren Verlauf dieses Kapitels werden die lexikalischen Elemente, die syntaktischen Elemente, der Präprozessor und die Standardbibliothek behandelt. Den Abschluss bilden besondere Kapitel zur Arbeitsweise eines C-Compilers sowie zu Codierungsregeln.

6.1 Lexikalische Elemente

Der Grundzeichensatz für C-Quelltexte umfasst folgende sichtbare Zeichen:

- Großbuchstaben: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- Kleinbuchstaben: a b c d e f g h i j k l m n o p q r s t u v w x y z
- Dezimalziffern: 0 1 2 3 4 5 6 7 8 9
- Unterstrich: _
- Interpunktion: ! " # % & ' ( ) * + , - . / : ; < = > ? [\ ] ^ { | } ~

Zusätzlich können folgende Zeichen vorkommen:

Zeichen	Bedeutung	Ersatzdarstellung
Space	Leerzeichen	
BEL	Alarmglocke (bell)	\a
BS	Rückschritt (backspace)	\b
FF	Seitenvorschub (form feed)	\f
NL	Zeilenvorschub (newline)	\n
CR	Wagenrücklauf (carriage return)	\r
HT	Horizontaltabulator (horizontal tab)	\t
VT	Vertikaltabulator (vertical tab)	\v

Es gibt auch Ersatzdarstellungen für die Anführungszeichen und zwei weitere Sonderzeichen zur Verwendung in Zeichen- und Zeichenkettenkonstanten. Hier dient der Rückschrägstrich dazu, die Sonderbedeutung des betr. Zeichens zu unterdrücken: \", \', \?, \\. Um alle Zeichen des Zeichensatzes der Maschine darstellen zu können, gibt es ferner so genannte numerische Escape-Sequenzen (Ersatzdarstellungen):

- \d, oder \dd oder \ddd d (1...3) ist Oktalziffer (oft gebraucht: '\0', die Null)
- \xh oder \xhh oder . . . h (beliebige Anzahl) ist Hexadezimalziffer (0 bis 9, A bis F oder a bis f)

In Zeichen- und Zeichenkettenkonstanten (auch Literale genannt) können alle Zeichen des verwendeten Systems vorkommen.

6.1.1 White Space (Leerraum)

Als Leerraum (white space) gelten Leerzeichen, Zeilenvorschub, Wagenrücklauf, vertikaler und horizontaler Tabulator, sowie Seitenvorschub. Kommentare gelten auch als Leerraum. Leerraum wird syntaktisch ignoriert, außer in Zeichenketten- oder Zeichenkonstanten; er dient dazu, sonst aneinandergrenzende Wörter, Zeichen etc. zu trennen und den Quelltext für Menschen durch übersichtliche Gestaltung, z.B. Einrückungen nach Kontrollstruktur etc., gut lesbar zu machen.

6.1.2 Kommentare

Kommentare werden durch die Zeichenpaare /* und */ erzeugt. Alles, was dazwischen steht – auf einer Zeile oder mit beliebig vielen Zeilen dazwischen, gilt als Kommentar. Kommentare dürfen nicht geschachtelt werden.

```
/* Das ist zum Beispiel ein Kommentar
. . . und hier geht er immer noch weiter */.
```

6.1.3 Schlüsselwörter

C hat die folgenden 32 Schlüsselwörter (reserved words, keywords):

```
auto break case char const continue default do
double else enum extern float for goto if int
long register return short signed sizeof static
struct switch typedef union unsigned void volatile
while
```

6.1.4 Identifier (Bezeichner)

Bezeichner in C (identifier), sonst auch schlicht Namen genannt, werden folgendermaßen gebildet (als regulärer Ausdruck in Unix-Notation):

[A-Za-z_][A-Za-z_0-9]*

d.h. Buchstabe oder Unterstrich optional gefolgt von beliebiger (auch Null) Folge eben dieser, inklusive der Ziffern.

Bezeichner dürfen nicht mit einer Ziffer beginnen, Groß- und Kleinbuchstaben sind als verschieden zu werten. Bezeichner dürfen nicht aus der Menge der o.g. Schlüsselwörter sein (oder aus der Menge von Namen, die für die Standardbibliothek reserviert sind, sie müssen sich mindestens in den ersten 31 Zeichen unterscheiden. Mit Unterstrich beginnende Namen sind für das System reserviert und sollten nicht verwendet werden. Bezeichner mit externer Bindung (d.h. Weiterverarbeitung durch Linker etc.) können weiteren Beschränkungen unterliegen.

6.1.5 Konstanten

C kennt vier Hauptgruppen von Konstanten:

- Ganzzahlkonstanten Dezimal-, Oktal- oder Hex-Darstellung
- Gleitpunktzahlkonstanten mit Dezimalpunkt und/oder Exponentkennung
- Zeichenkonstanten eingeschlossen in '. . . '
- Zeichenkettenkonstanten eingeschlossen in ". . . "

Numerische Konstanten sind immer positiv, ein etwa vorhandenes Vorzeichen gilt als unärer Operator auf der Konstanten und gehört nicht dazu. Ganzzahlkonstanten sind vom Typ int, wenn das nicht ausreicht, vom Typ long, wenn auch das nicht ausreicht, vom Typ unsigned long. Man kann die größeren Typen auch durch Anfügen von Suffixen erzwingen, wie aus der folgenden Tabelle ersichtlich. Beginnt die Ganzzahlkonstante mit 0x oder 0X, so liegt Hexnotation vor und es folgen eine oder mehrere Hexziffern. Dabei stehen A-F bzw. a-f für die Werte 10...15. Beginnt andernfalls die Ganzzahlkonstante mit einer 0, so liegt Oktalnotation vor und es

folgen eine oder mehrere Oktalziffern, andernfalls liegt Dezimalnotation vor. Gleitpunktzahlkonstanten sind immer vom Typ double, falls nicht durch Suffix als float oder long double gekennzeichnet. Zur Erkennung müssen mindestens der Dezimalpunkt oder die Exponentkennung vorhanden sein.

Dezimalziffern	0 1 2 3 4 5 6 7 8 9
Oktalziffern	0 1 2 3 4 5 6 7
Hexziffern	0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f
0	Die Konstante 0 (Null)
l L	Ganzzahlsuffix für long (Verwechselungsgefahr l mit 1!)
u U	Ganzzahlsuffix für unsigned
f F l L	Gleitpunktzahlsuffix für float bzw. long double (s.o.)
e E	Gleitpunktzahlkennung für Exponent

Tabelle 6.1 Darstellungen für Konstanten in C

Eine *Zeichenkonstante* (character constant) ist ein in einfache Hochkommata eingeschlossenes Zeichen aus dem Zeichensatz oder seine (auch mehrere Zeichen umfassende) Ersatzdarstellung. Die Betrachtung sog. wide character constants, sowie sog. multi byte character constants unterbleibt hier. Zeichenkonstanten sind vom Typ int, dürfen aber nicht wertmäßig größer als der entspr. Typ char sein.

Eine *Zeichenkettenkonstante* (string constant) ist eine in sog. doppelte Anführungszeichen eingeschlossene Zeichenkette auf einer Zeile. Sie darf alle Zeichen des Zeichensatzes, incl. etwaiger Ersatzdarstellungen, und (dann signifikanten) Leerraum enthalten. Nur durch Leerraum getrennte Zeichenketten werden vom Präprozessor zusammengefügt und gelten als eine Zeichenkette. Man kann eine Zeile auch umbrechen, indem man sie mit einem Rückschrägstrich terminiert. Die auf diese Weise fortgeführte Zeile gilt dann als eine logische Zeile.

Zeichenketten werden standardgemäß als array of char von niederen zu höheren Adressen mit terminierendem Nullwert im Speicher abgelegt. Ihre Speichergröße ist daher immer um 1 größer als die Größe, die der Anzahl der enthaltenen Zeichen entsprechen würde. Das sind also die allseits verbreiteten so genannten C-Strings. Der Nullwert dient als Terminierungsmarke für alle Routinen der Standardbibliothek und kann folglich im String selbst nicht vorkommen. Der terminierende Nullwert gehört somit nicht zu den Zeichen des Strings und wird folglich bei Ermittlung seiner Länge auch nicht mitgezählt.

Eine Zeichenkette als Typ array of char zu sehen, nimmt man aber nur bei der Initialisierung von Arrays oder der Anwendung des `sizeof`-Operators wahr. Bei den meisten Verwendungen treten jedoch sofort die üblichen syntaktischen Umwandlungen von C in Kraft, und man sieht nur noch einen Zeiger auf das erste Zeichen, also den Typ char *, über den man dann alle weitere Verarbeitung steuern kann.

6.2 Syntaktische Elemente

6.2.1 Datentypen

Der Begriff des Datentyps beinhaltet folgendes:

- die Größe und Ausrichtung des belegten Speicherplatzes (size, alignment)
- die interne Darstellung (Bitbelegung)
- die Interpretation und damit den Wertebereich
- die darauf anwendbaren bzw. erlaubten Operationen

C-Typbezeichnung		Gruppe	Klasse	Kategorie	MinBit
char		integer	arithmetic	scalar	8
signed char		integer	arithmetic	scalar	8
unsigned char		integer	arithmetic	scalar	8
short, signed short		integer	arithmetic	scalar	16
unsigned short		integer	arithmetic	scalar	16
int, signed int		integer	arithmetic	scalar	16
unsigned int		integer	arithmetic	scalar	16
long, signed long		integer	arithmetic	scalar	32
unsigned long		integer	arithmetic	scalar	32
enum		integer	arithmetic	scalar	s.d.
float		float	arithmetic	scalar	(32)
double		float	arithmetic	scalar	s.w.u.
long double		float	arithmetic	scalar	s.w.u.
T *	(pointer to T)		pointer	scalar	
T [...]	(array of T)		array	aggregate	
struct {...}			struct	aggregate	
union {...}			union	aggregate	
T (...)	(function returning T)			function	
void				void	

Tabelle 6.2 Übersicht zu den intrinsischen Datentypen in C

ISO-C verfügt über einen reichhaltigen Satz von Datentypen, die sich wie in vorangegangener Übersicht gezeigt organisieren lassen. ISO-C verlangt binäre Codierung der integralen Typen. Für die Wertebereiche aller arithmetischen Typen

sind Mindestwerte und Größenverhältnisse festgelegt. Die implementierten Größen dieser Datentypen sind in `limits.h` und `float.h` definiert.

In obiger Tabelle bezeichnet T* einen Zeiger auf den Typ T, T[...] ein Array vom Typ T, T(...) eine Funktion mit Rückgabetyp T. void ist der leere Typ. Als Rückgabetyp einer Funktion deklariert zeigt er an, dass die Funktion nichts zurückgibt, in der Parameterliste, dass sie nichts nimmt. Ein Zeiger auf void ist ein Zeiger auf irgendetwas unbestimmtes, ein generischer Zeiger, den man nie dereferenzieren kann. Variablen oder Arrays vom Typ void können daher nicht deklariert werden. Der Array-Typ T[] und der Funktionstyp T() können nicht Typ einer Funktion sein.

Die Gruppen, Klassen und Kategorien dienen zur Kenntlichmachung der auf diesen Typen und in Verbindung mit diesen Typen erlaubten Operationen. Datentypen können durch die sog. type qualifiers `const` und `volatile` weiter qualifiziert werden. Dabei bedeutet `const`, dass ein so bezeichneter Datentyp nur gelesen werden darf (read only), d.h. er könnte z.B. in einem solchen Speicherbereich oder im ROM abgelegt sein. `volatile` bedeutet, dass die so qualifizierte Größe durch außerhalb des Wirkungsbereichs des Compilers liegende Einflüsse verändert werden könnte, z.B. kann es sich hier um in den Speicherbereich eingeblendete Hardwareregister (sog. Ports) handeln. Dies soll den Compiler davon abhalten, gewisse sonst mögliche Optimierungen des Zugriffs auf die entsprechende Variable vorzunehmen. Beide Qualifizierer können auch zusammen auftreten. Hier einige Beispiele:

```
int i; /* i ist als Variable vom Typ int definiert */
const int ic = 4711; /* ic ist als Konstante vom Typ int
                        definiert */
const int *pc; /* pc ist Zeiger auf konstanten int */
int *const cpi = &i; /* cpi ist konstanter Pointer auf int */
const int *const cpc = &ic; /* konstanter Pointer auf
                               konstanten int */
volatile int vi; /* vi kann durch äußeren Einfluss verändert
                    werden */
const volatile int vci; /* vci ist z.B. ein Timerport */
```

Als `const` vereinbarte Variablen dürfen vom Programm nicht verändert werden. Falls man es versucht, gibt es Fehlermeldungen vom Compiler. Falls man dies jedoch durch in C legale Mittel wie Typumwandlung zu umgehen versucht, kann es je nach System auch zu Laufzeitfehlern führen.

6.2.2 Deklarationen und Definitionen

C ist eine eingeschränkt blockstrukturierte Sprache, d.h. Blöcke sind das strukturelle Gliederungsmittel. Blöcke werden durch die Blockanweisung { ... } erzeugt. Die Einschränkung ist, dass Funktionsdefinitionen (siehe dort) nur außerhalb von Blöcken möglich sind. Blöcke können beliebig geschachtelt werden. Alles, was außerhalb von Blöcken deklariert oder definiert wird, ist global. Alles, was in ei-

nem Block deklariert oder definiert wird, ist lokal zu diesem Block und gilt bis zum Verlassen dieses Blocks. Ein in einem Block deklarierter Name kann einen in einer höheren Ebene deklarierten Namen maskieren, d.h. der äußere Name wird verdeckt und das damit bezeichnete Objekt ist dort nicht mehr zugreifbar.

Der Compiler bearbeitet (man sagt auch liest) den Quelltext (genauer die vom Präprozessor vorverarbeitete Übersetzungseinheit) Zeile für Zeile, von links nach rechts und von oben nach unten. Bezeichner (Identifier, → 6.1.4) müssen grundsätzlich erst eingeführt sein, d.h. deklariert und/oder definiert sein, bevor sie benutzt werden können.

Deklarationen machen dem Compiler Bezeichner (Namen) und ihren Typ bekannt. Sie können auch unvollständig sein, d.h. nur den Namen und seine Zugehörigkeit zu einer bestimmten Klasse bekannt machen, ohne wissen zu müssen, wie der Typ nun genau aussieht. Das reicht dann nicht aus, um dafür Speicherplatz zu reservieren, aber man kann z.B. einen Zeiger auf diesen jetzt noch unvollständigen Typ erzeugen, um ihn dann später, wenn der Typ vollständig bekannt ist, auch zu benutzen. Deklarationen können, abhängig von ihrer Typklasse, auch Definitionen sein. Wenn sie global, d.h. außerhalb von Blöcken erfolgen, sind sie standardmäßig auf den Wert Null initialisiert. Innerhalb eines Blocks ist ihr Wert bei ausbleibender Initialisierung undefiniert. Definitionen haben die Form:

`Typ Name;` oder `Typ Name1 ,Name2, . . . ;`

Definitionen weisen den Compiler an, Speicherplatz bereitzustellen und, wenn das angegeben wird, mit einem bestimmten Wert zu initialisieren. Eine Definition ist gleichzeitig auch eine Deklaration. Eine Definition macht den Typ vollständig bekannt und benutzbar, d.h. es wird Speicherplatz dafür reserviert (im Falle von Datentypen) oder auch Code erzeugt (im Falle von Funktionsdefinitionen, siehe dort).

Definitionen von Datenobjekten mit Initialisierung haben die Form:

`Typ Name =Wert;` oder `Typ Name1 =Wert1 ,Name2 =Wert2 ,. . . ;`

6.2.3 Speicherklassen, Sichtbarkeit und Bindung

Außerhalb von Blöcken vereinbarte Objekte gehören zur Speicherklasse `static`. Sie sind vom Start des Programms an vorhanden, und sind global, d.h. im ganzen Programm gültig und sichtbar – sie haben *global scope* und externe Bindung (*external linkage*). Wenn sie nicht im Programm auf bestimmte Werte gesetzt sind, werden sie auf den Wert 0 initialisiert (im Gegensatz zur Speicherklasse auto).

Durch Angabe des Schlüsselworts static kann der Sichtbarkeitsbereich (*scope*) für so vereinbarte Objekte auf die Übersetzungseinheit (Datei) eingeengt werden, das Objekt hat dann interne Bindung (*internal linkage*) und *file scope*.

Deklarationen und Definitionen in Blöcken können nur vor allen Anweisungen (siehe dort) stehen, also zu Beginn eines Blocks. Sie sind lokal zu dem Block, in

dem sie erscheinen (*block scope*). Die so vereinbarten Objekte haben die Speicherklasse `auto`, d.h. sie existieren nur, solange der Block aktiv ist und werden bei Eintritt in den Block jedes Mal wieder neu erzeugt, jedoch ohne definierten Anfangswert. Durch Angabe des Schlüsselworts `static` kann die Speicherklasse auf `static` geändert werden, ohne dass Sichtbarkeit und Bindung davon berührt würden. Sie sind von Beginn an vorhanden und behalten ihren Wert auch nach Verlassen des Gültigkeitsbereichs.

Vereinbarungen von Objekten mittels `register` sind nur in Blöcken oder in Parameterlisten von Funktionen erlaubt und dienen lediglich als Hinweis an den Compiler, er möge sie in (schnellen) Prozessorregistern ablegen. Ob das denn auch geschieht, bleibt dem Compiler überlassen. Auf so vereinbarte Objekte darf der Adressoperator & nicht angewandt werden.

Der Sichtbarkeitsbereich einer Marke (label) ist die Funktion, in der sie deklariert ist (*function scope*). Innerhalb einer Funktion weist man bei Vereinbarung eines Namens mit extern darauf hin, dass das Objekt anderweitig definiert ist. Außerhalb von Funktionen gelten alle vereinbarten Objekte defaultmäßig als `extern`.

6.2.4 Operatoren

C verfügt über einen reichhaltigen Satz von Operatoren. Diese lassen sich nach verschiedenen Kategorien gliedern:

- nach der Art: unäre, binäre und ternäre Operatoren
- nach Vorrang – Präzedenz (precedence)
- nach Gruppierung – Assoziativität: links, rechts (associativity)
- nach Stellung: Präfix, Infix, Postfix
- nach Darstellung: einfach, zusammengesetzt

Die Vielfalt und oft mehrfache Ausnutzung der Operatorzeichen auch in anderem syntaktischen Zusammenhang bietet anfangs ein verwirrendes Bild. Der Compiler kann aber immer nach dem Kontext entscheiden, welche der Operatorfunktionen gerade gemeint ist. Zur Erleichterung des Verständnisses der Benutzung und Funktionsweise der Operatoren daher folgend einige Anmerkungen zur Operatortabelle.

`[]` → 6.2.9 Vektoren und Zeiger.

`*` und `->` → 6.2.9 Vektoren und Zeiger

`++`, `--` (z.B. a++, b--) als Postin- bzw. -dekrement liefern sie den ursprünglichen Wert ihres Operanden und erhöhen bzw. erniedrigen den Wert des Operanden danach um 1. Diese Operatoren können nur auf Objekte im Speicher angewandt werden, die vom skalaren Typ sein müssen und auf die schreibend zugegriffen werden kann. Wann die tatsächliche Veränderung des Operandenwertes im Speicher eintritt, der Seiteneffekt dieser Operatoren, ist implementationsabhängig und erst nach dem Passieren eines Sequenzpunktes sicher.

++, - - (z.B. ++a, --b) als Präin- bzw. -dekrement erhöhen bzw. erniedrigen sie erst den Wert ihres Operanden um 1 und liefern dann den neuen, so erhöhten Wert.

C-Bezeichnung	Erläuterung d. Funktion	Klasse	Vorrang	Gruppierung
[]	Indexoperator	postfix	16	links
()	Funktionsaufruf	postfix	16	links
.	direkte Komponentenwahl	postfix	16	links
->	indir. Komponentenwahl	postfix	16	links
++ --	Postinkrement, -dekrement	postfix	16	links
++ --	Präinkrement, -dekrement	präfix	15	rechts
sizeof	Größe ermitteln	unär	15	rechts
~	bitweise Negation	unär	15	rechts
!	logische Negation	unär	15	rechts
- +	arithm. Negation, plus	unär	15	rechts
&	Adresse von	unär	15	rechts
*	Indirektion	unär	15	rechts
(type name)	Typumwandlung (cast)	unär	14	rechts
* / %	mult., div., mod.	binär	13	links
+ -	Addition, Subtraktion	binär	12	links
<< >>	bitweise schieben	binär	11	links
< > <= >=	Vergleiche	binär	10	links
== !=	gleich, ungleich	binär	9	links
&	bitweises AND	binär	8	links
^	bitweises XOR	binär	7	links
|	bitweises OR	binär	6	links
&&	logisches AND	binär	5	links
||	logisches OR	binär	4	links
? :	Bedingungsoperator	ternär	3	rechts
=	Zuweisung	binär	2	rechts
+= -= *= /= %=	Verbundzuweisung	binär	2	rechts
<<= >>= &= ^= |=	Verbundzuweisung	binär	2	rechts
,	Sequenzoperator	binär	1	links

Tabelle 6.3 Operatoren in C

Diese Operatoren können nur auf Objekte im Speicher angewandt werden, die vom skalaren Typ sein müssen und auf die schreibend zugegriffen werden kann. Wann die tatsächliche Veränderung des Operandenwertes im Speicher eintritt, der *Seiteneffekt* dieser Operatoren, ist implementationsabhängig und erst nach dem Passieren eines Sequenzpunktes sicher.

`sizeof` dieser Operator arbeitet zur Compilierungszeit (sog. *compile time operator*) und liefert die Größe seines Operanden in Einheiten des Typs `char`: `sizeof(char)` == 1. Der Operand kann ein Typ sein, dann muss er in () stehen, oder ein Objekt im Speicher, dann sind keine Klammern erforderlich. Ist der Operand ein Arrayname, liefert er die Größe des Arrays in `char`-Einheiten.

~ (z.B. ~a): die Tilde liefert den Wert der bitweisen Negation (das Komplement) der Bitbelegung ihres Operanden, der vom integralen Typ sein muss.

! (z.B. !a): liefert die logische Negation des Wertes seines Operanden, der vom skalaren Typ sein muss. War der Wert 0, ist das Ergebnis 1, war der Wert ungleich 0 , ist das Ergebnis 0.

`-`, `+` (z.B. -a): die unäre Negation liefert den negierten Wert ihres Operanden, der vom arithmetischen Typ sein muss. Das unäre Plus wurde nur aus Symmetriegründen eingeführt, und dient evtl. lediglich Dokumentationszwecken.

& (z.B. &a) _ liefert die Adresse eines Objektes im Speicher (und erzeugt somit einen Zeigerausdruck, → 5.2.9).

* (z.B. *a): in einer Deklaration erzeugt dieser Operator einen Zeiger auf den deklarierten Typ, in der Anwendung auf einen Zeigerwert, liefert er den Wert des so bezeigten Objekts (→ 5.2.9).

(typename): *typename* ist ein Typbezeichner. Der sog. *type cast operator* liefert den in diesen Typ konvertierten Wert seines Operanden. Dabei wird versucht, den Wert zu erhalten. Eine (unvermeidbare, beabsichtigte) Wertänderung tritt ein, wenn der Wert des Operanden im Zieltyp nicht darstellbar ist, ähnlich einer Zuweisung an ein Objekt dieses Typs. Im Folgenden einige Hinweise zu erlaubten Konversionen:

- Jeder arithmetische Typ in jeden arithmetischen Typ.
- Jeder Zeiger auf void in jeden Objektzeigertyp.
- Jeder Objektzeigertyp in Zeiger auf void.
- Jeder Zeiger auf ein Objekt oder void in einen Integertyp.
- Jeder Integertyp in einen Zeiger auf ein Objekt oder void.
- Jeder Funktionszeiger in einen anderen Funktionszeiger.
- Jeder Funktionszeiger in einen Integertyp.
- Jeder Integertyp in einen Funktionszeiger.

Die Zuweisung von `void`-Zeiger an Objektzeiger und umgekehrt geht übrigens auch ohne den Typkonversionsoperator. In allen anderen Fällen ist seine Anwendung geboten oder erforderlich, und sei es nur, um den Warnungen des Compilers zu entgehen.

% (z.B. a%b): modulo liefert den ganzzahligen Divisionsrest des Wertes seines linken Operanden geteilt durch den Wert seines rechten Operanden und lässt

sich nur auf integrale Typen anwenden. Dabei sollte man Überlauf und die Division durch Null vermeiden. Bei positiven Operanden wird der Quotient nach 0 abgeschnitten. Falls negative Operanden beteiligt sind, ist das Ergebnis implementationsabhängig. Es gilt jedoch immer: X = (X/Y) * Y + (X%Y).

Die übrigen arithmetischen Binäroperatoren können nur auf Operandenpaare vom arithmetischen Typ angewandt werden, dabei geht, wie üblich und auch aus der Tabelle zu ersehen, Punktrechnung vor Strichrechnung. Bei der Ganzzahldivision wird ein positiver Quotient nach 0 abgeschnitten. Man vermeide auch hier die Null als Divisor. Wenn unterschiedliche Typen an der Operation beteiligt sind, wird selbstständig in den größeren der beteiligten Typen umgewandelt (balanciert).

<<, >> (z.B. a<<b): die Bitschiebeoperatoren schieben den Wert des linken Operanden um Bitpositionen des Wertes des rechten Operanden jeweils nach links bzw. rechts und können nur auf integrale Operandenpaare angewandt werden. Für eine n-Bit-Darstellung des promovierten linken Operanden muss der Wert des rechten Operanden im Intervall 0..n-1 liegen. Bei positivem linken Operanden werden Nullen in die freigewordenen Positionen nachgeschoben. Ob bei negativem linken Operanden beim Rechtsschieben das Vorzeichen nachgeschoben wird (meist so gehandhabt), oder Nullen, ist implementationsabhängig.

Die Vergleichsoperatoren (z.B. a == b) können nur auf arithmetische und auf Paare von Zeigern gleichen Typs angewandt werden. Sie liefern den Wert 1, wenn der Vergleich erfolgreich war, sonst 0.

Die bitlogischen Operatoren (z.B. a&b) können nur auf integrale Typen angewandt werden und liefern den Wert der bitlogischen Verknüpfung des Wertes des linken mit dem Wert des rechten Operanden (beide als Bitmuster interpretiert).

&& (z.B. a && b): testet, ob beide Operanden ungleich Null (wahr) sind. Ist der linke Operand wahr, wird auch der rechte getestet, andernfalls hört man auf, und der rechte Operand wird nicht mehr bewertet, da das Ergebnis der logischen UND-Verknüpfung ja schon feststeht (sog. Kurzschlussbewertung, *short circuit evaluation*, mit Sequenzpunkt nach dem linken Operanden). Beide Operanden müssen vom skalaren Typ sein. Im Wahrheitsfall ist der Wert des Ausdrucks 1, sonst 0.

|| (z.B. a || b): testet, ob mindestens einer der beiden Operanden ungleich Null (wahr) ist. Ist der linke Operand gleich Null (falsch), wird auch der rechte getestet, andernfalls hört man auf, und der rechte Operand wird nicht mehr bewertet, da das Ergebnis der logischen ODER-Verknüpfung ja schon feststeht (sog. Kurzschlussbewertung, short circuit evaluation, wie oben) Beide Operanden müssen vom skalaren Typ sein. Im Wahrheitsfall ist der Wert des Ausdrucks 1, sonst 0.

`X?Y:Z` X muss vom skalaren Typ sein und wird bewertet. Ist X ungleich Null (wahr), wird Y bewertet, andernfalls wird Z bewertet. Y und Z können fast beliebige Ausdrücke sein, auch `void` ist möglich, sollten aber kompatibel sein. Zwischen der Bewertung von X und der Bewertung von entweder Y oder Z befindet sich ein Sequenzpunkt (sequence point). Der Wert des Ausdrucks ist dann der Wert des (evtl. im Typ balancierten) Wertes des zuletzt bewerteten Ausdrucks.

= Der Zuweisungsoperator bewertet seine beiden Operanden von rechts nach links, so sind auch Zuweisungsketten in der Art von a = b = c = d = 4711 möglich. Der Wert des Zuweisungsausdrucks ist der Wert des Zugewiesenen, der in den Typ des linken Operanden transformierte Wert des rechten Operanden. Der linke Operand muss ein Objekt im Speicher darstellen, auf das schreibend zugegriffen werden kann. Aufgrund der speziellen Eigenheit von C, dass die Zuweisung ein Ausdruck und keine Anweisung ist, sowie seiner einfachen Wahr-Falsch-Logik, taucht die Zuweisung oft als Testausdruck zur Schleifenkontrolle auf. Ein markantes Beispiel:

```
while (*s++ = *t++); /* C-Idiom für Zeichenketten
                        kopie */
```

Die Verbund- oder Kombinationszuweiser bestehen aus zwei Zeichen, deren rechtes der Zuweiser ist. Sie führen, kombiniert mit der Zuweisung verschiedene arithmetische, bitschiebende und bitlogische Operationen aus. Dabei bedeutet a op= b soviel wie a = a op b, mit dem Unterschied, dass a, also der linke Operand, nur einmal bewertet wird.

Der Komma- oder Sequenzoperator (z.B. a,b) gruppiert wieder von links nach rechts und bewertet erst seinen linken, dann seinen rechten Operanden. Dazwischen liegt ein Sequenzpunkt, das heißt, alle Seiteneffekte sind garantiert eingetreten. Der Wert des Ausdrucks ist das Resultat der Bewertung des rechten Operanden. Der Nutzen des Operators besteht darin, dass er einen Ausdruck erzeugt und folglich überall stehen kann, wo ein Ausdruck gebraucht wird. Seine Hauptanwendungen sind die Initialisierungs- und Reinitialisierungsausdrücke in der Kontrollstruktur der `for`-Schleife, wo ja jeweils nur ein Ausdruck erlaubt ist, und manchmal mehrere gebraucht werden.

Einige Operationen erzeugen implementationsabhängige Typen, die in `stddef.h` definiert sind. `size_t` ist der vom `sizeof`-Operator erzeugte vorzeichenlose integrale Typ. `ptrdiff_t` ist der vorzeichenbehaftete integrale Typ, der vom Subtraktionsoperator erzeugt wird, wenn dieser auf Zeiger (gleichen Typs!) angewandt wird.

6.2.5 Ausdrücke

C ist eine Ausdrucks-orientierte Sprache. Der Compiler betrachtet die Ausdrücke und bewertet sie. Ein Ausdruck (*expression*) in C ist:

- eine Konstante (constant)
- eine Variable (variable)
- ein Funktionsaufruf (function call)
- eine beliebige Kombination der obigen 3 Elemente mittels Operatoren

Jeder Ausdruck hat einen Typ und einen Wert. Bei der Bewertung von Ausdrücken gelten folgende Regeln: Daten vom Typ `char` oder `short` werden sofort in den Typ `int` umgewandelt (*integral promotion*). Bei der Kombination von Ausdrücken wird balanciert, d.h. der dem Wertebereich oder Speicherplatz nach kleinere Typ wird in den beteiligten, dem Wertebereich oder Speicherplatz nach größeren Typ umgewandelt. Dabei wird versucht, den Wert zu erhalten (*value preservation*).

Die Bewertung der einzelnen Elemente von Ausdrücken folgt Vorrang und Assoziativität der Operatoren. Bei Gleichheit in diesen Eigenschaften ist die Reihenfolge der Bewertung (*order of evaluation*) gleichwohl bis auf wenige Ausnahmen undefiniert, denn der Compiler darf sie auf für ihn günstige Weise ändern, wenn das Ergebnis aufgrund der üblichen mathematischen Regeln gleichwertig wäre. In der Theorie gilt (a * b)/c = (a/c) * b, also darf der Compiler das nach seinem Gusto umordnen, und auch Gruppierungsklammern können ihn nicht daran hindern. Das kann aber bei den Darstellungs-begrenzten Datentypen im Computer schon zu unerwünschtem Überlauf etc. führen.

Soll dies wirklich verhindert werden, d.h., soll der Compiler gezwungen werden, eine bestimmte Reihenfolge einzuhalten, muss die entsprechende Rechnung aufgebrochen und in mehreren Teilen implementiert werden. Die Codesequenzen

```
x = (a * b) / c;
```

und

```
x = a * b;
x = x / c;
```

bewirken tatsächlich nicht automatisch das gleiche, denn im ersten Fall darf der Compiler umsortieren, im zweiten nicht, da das Semikolon einen so genannten Sequenzpunkt (*sequence point*) darstellt, den der Compiler nicht entfernen darf.

Manche Operatoren bewirken sog. Seiteneffekte (side effects), d.h. sie können den Zustand des Rechners verändern, z.B. den Wert von Speichervariablen oder Registern oder sonstiger Peripherie. Dazu gehören neben den Zuweisern auch die Post- und Präinkrement und -dekrement-Operatoren und Funktionsaufrufe. Das Eintreten der Wirkung dieser Seiteneffekte sollte niemals abhängig von der Reihenfolge der Bewertung sein! Während durch Komma separierte Deklarations- und Definitionslisten strikt von links nach rechts abgearbeitet und bewertet werden, gilt das z.B. für die Reihenfolge der Bewertung in Parameterlisten beim Funktionsaufruf nicht.

6.2.6 Anweisungen

In C gibt es folgende Anweisungen (*statements*):

- Leeranweisung `;` (*empty statement*)
- Audrucksanweisung *expression*`;` (*expression statement*)
- Blockanweisung `{ ... }` (*block statement*)
- markierte Anweisung *label*`:` *statement* (*labeled statement*)
- Auswahlanweisung `if else switch ... case` (*selection statement*)
- Wiederholungsanweisung `for while do ... while` (*iteration statement*)
- Sprunganweisung `goto break continue return` (*jump statement*)

6.2.7 Kontrollstrukturen

Kontrollstrukturen definieren den Ablauf eines Programms. Die einfachste Kontrollstruktur ist die *Sequenz*, d.h. Folge. Der Compiler liest den Quelltext von links nach rechts, von oben nach unten, und setzt ihn in Code um, der eine sequentielle Abarbeitung bewirkt. Um dies zu erzeugen, schreibt man also eine Anweisung nach der andern, von links nach rechts, bzw. besser von oben nach unten, hin.

Die nächste Kontrollstruktur ist die *Auswahl*. C kennt die zwei Auswahl- oder Verzweigungskontrollstrukturen `if else` und `switch case`. Das `if`-Konstrukt hat folgende allgemeine Form:

```
if (expression) /* expression muss vom arithme
                   tischen oder Zeigertyp sein */
   statement1   /* wenn expression ungleich 0,
                   statement1 ausführen */
else
   statement2   /* sonst statement2 ausführen */
```

Der `else`-Teil ist optional. Man beachte, dass es in C kein *then* und keine Endmarke (*endif* o.ä.) für diese Konstruktion gibt. Ebenso ist jeweils nur ein *statement* erlaubt; braucht man mehrere, so muss man zum {*block statement*} greifen. Falls man mehrere geschachtelte `if`-Strukturen verwendet, ordnet der Compiler das `else` immer dem jeweilig direkt vorausgehenden `if` zu, so dass man durch Verwendung von Blockklammern `{}` für die korrekte Gliederung sorgen muss, die visuelle Gestaltung des Quelltexts ist nur eine Lesehilfe und hat für die Syntax der Sprache C keine Bedeutung.

Das zweite Auswahlkonstrukt, `switch case`, hat viele Formen, am häufigsten gebraucht wird die folgende allgemeine Form:

```
switch (integral expression) {
  case constintexpr1 : /* Der : ist die Syntaxken
                          nung für eine Marke. */
     statement1
```

```
      statement2
      break; /* hier wird der switch in diesem Fall
                verlassen. */
    case constintexpr2 :
      statement3
      statement4  /* break fehlt: Es geht weiter zum
                     nächsten Fall! */
    default:
      statement5
  }
```

Die Ausdrücke in den `case`-Marken müssen konstante integrale Ausdrücke sein. Mehrere Marken sind erlaubt. Für den kontrollierenden Ausdruck findet Integer-Erweiterung statt und die `case`-Konstanten werden in den so erweiterten Typ umgewandelt. Danach dürfen keine zwei Konstanten den gleichen Wert haben. Die `default`-Marke darf pro `switch` nur einmal vorhanden sein; sie deckt alles ab, was von den anderen Marken nicht erfasst wird und darf an beliebiger Stelle erscheinen.

Das Problem des `switch`-Konstrukts ist die `break`-Anweisung: fehlt sie, geht die Abarbeitung über die nächste Marke hinweg einfach weiter (sog. *fall through*). Dies kann man natürlich geschickt ausnutzen, ein fehlendes – vergessenes – `break` hat jedoch oft schon zu den seltsamsten Überraschungen geführt. Es ist daher zu empfehlen, einen beabsichtigten Fall von *fall through* durch entspr. Kommentar besonders kenntlich zu machen.

Die nächste wichtige Kontrollstruktur ist die Wiederholung, auch Schleife genannt. Hier hält C drei verschiedene Konstrukte bereit:

```
while (expression) /*solange expression ungleich 0 */
   statement       /* statement ausführen */
```

expression muss vom arithmetischen oder Zeigertyp sein und wird bewertet. Falls nicht 0, wird *statement* ausgeführt; dies wird solange wiederholt, bis *expression* zu 0 bewertet wird. Dies ist eine sog. kopfgesteuerte Schleife. Soll das `while`-Konstrukt mehrere Anweisungen kontrollieren, greift man üblicherweise zur Blockanweisung.

```
do
   statement         /* statement ausführen */
while (expression); /* solange bis expression zu 0
                       bewertet wird */
```

statement wird ausgeführt, dann wird *expression* bewertet. *expression* muss wie oben vom arithmetischen oder Zeigertyp sein. Falls nicht 0, wird dies solange wiederholt, bis *expression* zu 0 bewertet wird. Man beachte das syntaktisch notwen-

dige Semikolon am Schluss des Konstrukts. Dies ist eine sog. fussgesteuerte Schleife. Für mehrere zu kontrollierende Anweisungen gilt das gleiche wie oben.

```
for (expression1; expression2; expression3)
   statement
```

Jeder der drei Ausdrücke in der Klammer des `for`-Konstrukts darf auch fehlen, die beiden Semikola sind jedoch syntaktisch notwendig. Zu Beginn wird einmalig *expression1* bewertet, ihr Typ unterliegt keiner Einschränkung. Sind mehrere Ausdrücke erforderlich, ist dies der Platz für den Einsatz des Sequenzoperators (,). Hier erfolgt daher meist die Initialisierung der Schleife. Als nächstes wird *expression2* bewertet, sie muss vom arithmetischen oder Zeigertyp sein. Ist der Wert ungleich 0, so wird *statement* ausgeführt. Alsdann wird *expression3* bewertet, ihr Typ unterliegt keiner Einschränkung. Hier erfolgt meist die Reinitialisierung der Schleife. Dann wird wieder *expression2* bewertet. Der Zyklus wird solange wiederholt, bis die Bewertung von *expression2* 0 ergibt. Fehlt *expression2*, wird dieser Fall als ungleich 0 bewertet.

Die `for`-Schleife ist gut lesbar und übersichtlich, da Initialisierung, Test und Reinitialisierung dicht beieinander und sofort im Blickfeld sind, bevor man überhaupt mit der Betrachtung des Schleifenkörpers beginnt. Sie ist daher sehr beliebt.

So genannte Endlosschleifen formuliert man in C folgendermaßen:

```
for (;;) statement
```

oder

```
while (1) statement
```

Den letzten Teil der Kontrollstrukturen bilden die sog. Sprunganweisungen:

`goto label;` springt zu einer Marke in der umgebenden Funktion. Diese Anweisung findet in der strukturierten Programmierung keine Verwendung und wird auch im Systembereich nur selten gebraucht. Sie ist jedoch nützlich in der (nicht für menschliche Leser bestimmten) maschinellen Codegenerierung.

`break;` darf nur in `switch` oder in Wiederholungsanweisungen stehen und bricht aus der es umgebenden Anweisung aus.

`continue;` darf nur in Wiederholungsanweisungen stehen und setzt die es umgebende Anweisung am Punkte der Wiederholung fort.

`return expressionopt` ; kehrt aus einer umgebenden Funktion mit der optionalen *expression* als Rückgabewert zurück.

6.2.8 Funktionen

Funktionen sind das Hauptgliederungsmittel eines Programms. Jedes gültige C-Programm muss eine bestimmte Funktion enthalten, nämlich die Funktion `main()`.

Funktionen in C erfüllen die Aufgaben, die in anderen Programmiersprachen *function*, *procedure* oder *subroutine* genannt werden. Sie dienen dazu, die Aufgaben des Programms in kleinere, übersichtliche Einheiten mit klaren und wohldefinierten Schnittstellen zu unterteilen. Funktionsdeklarationen haben die allgemeine Form:

```
Typ Funktionsname(Parameterliste);
```

Wird Typ nicht angegeben, so wird `int` angenommen, man sollte dies aber unbedingt vermeiden. Ist die Parameterliste leer, kann die Funktion eine unspezifizierte Anzahl (auch Null) Parameter unspezifizierten Typs nehmen. Besteht die Parameterliste nur aus dem Schlüsselwort `void`, nimmt die Funktion keine Parameter. Andernfalls enthält die Parameterliste einen oder mehrere Typnamen, optional gefolgt von Variablennamen, als durch Komma separierte Liste.

Als letzter (von mindestens zwei) Parametern ist als Besonderheit auch die Ellipse (…) erlaubt und bedeutet dann eine variable Anzahl sonst unspezifizierter Parameter.

Die Variablennamen haben für den Compiler keine Bedeutung, können aber dem Programmierer als Hinweis auf die beabsichtigte Verwendung dienen, im Sinne einer besseren Dokumentation. Zum Beispiel sind diese beiden Deklarationen,

```
int myfunc(int length, int count, double factor);
```

oder

```
int myfunc(int, int, double);
```

auch *Funktionsprototypen* genannt, für den Compiler identisch. Die Typangaben der Parameterliste, ihre Anzahl und Reihenfolge – auch Signatur (*signature*) genannt – dienen dem Compiler zur Fehlerdiagnose beim Aufruf, d.h. der Benutzung der Funktion. Deshalb sollten Funktionsdeklarationen – die Prototypen – der Benutzung der Funktionen – dem Funktionsaufruf (*function call*) – immer vorausgehen.

Anmerkung: Da die Variablennamen in der Wahl frei sind (wobei sie natürlich den syntaktischen Bedingungen genügen müssen), können sie auch im Namen den Variablentyp in codierter Form mitführen. So erweist sich die Deklaration

```
int iMyfunc(int iLength, int iCount, double dfFactor);
```

als besonders gut lesbar in dem Sinn, dass mitten im Text aus dem Variablennamen auch auf den Typ geschlossen werden kann.

Funktionsdefinitionen haben die allgemeine Form:

```
Typ Funktionsname(Parameterliste)
{
   Deklarationen und Definitionen
   Anweisungen
}
```

Funktionen können nur außerhalb von Blöcken definiert werden. Eine Funktionsdefinition ist immer auch gleichzeitig eine Deklaration. Der Hauptblock einer Funktion, auch Funktionskörper genannt, ist der einzige Ort, wo Code (im Sinne von ausführbaren Prozessorbefehlen) erzeugt werden kann.

Typ und Signatur einer Funktion müssen mit etwaigen vorausgegangenen Prototypen übereinstimmen, sonst gibt es Fehlermeldungen vom Compiler. Beim Funktionsaufruf (*function call*) schreibt man lediglich den Namen der Funktion, gefolgt von den in Klammern gesetzten Argumenten, oft auch aktuelle Parameter genannt, als durch Komma separierte Liste. Die Argumente nehmen den Platz der formalen Parameter ein und werden, da dies ein Zuweisungskontext ist, im Typ angeglichen. Die Reihenfolge der Bewertung dieser Argumentzuweisung ist dabei nicht festgelegt – es ist nur sichergestellt, dass alle Argumente bewertet sind, bevor der eigentliche Aufruf, d.h. der Sprung zum Code des Funktionskörpers erfolgt.

Falls die Funktion ein Ergebnis liefert, den sog. Funktionswert, kann man dieses zuweisen oder weiter verarbeiten, muss es aber nicht (wenn man z.B. nur an dem Seiteneffekt interessiert ist). Ein Beispiel dazu:

```
len = strlen("hello, world\n"); /* Funktionswert
                                   zuweisen */
printf("hello, world\n"); /* kein Interesse am
                             Funktionswert */
```

Ein Funktionsaufruf stellt einen Ausdruck dar und darf überall stehen, wo ein Ausdruck des Funktionstyps stehen kann. Eine `void`-Funktion hat definitionsgemäß keinen Funktionswert und ihr Aufruf darf daher nur in einem für diesen Fall zulässigen Zusammenhang erscheinen (z.B. nicht in Zuweisungen, Tests etc.).

Die Ausführung des Funktionskörpers endet mit einer return-Anweisung mit einem entspr. Ausdruck, dies ist wieder als Zuweisungskontext zu betrachten, und es wird in den Typ der Funktion konvertiert. Eine `void`-Funktion endet mit einer ausdruckslosen `return`-Anweisung oder implizit an der endenden Klammer des Funktionsblocks.

Die Funktion main()

Die Funktion `main()` spielt eine besondere Rolle in der Sprache C. Ihre Form ist vom System vordefiniert, sie wird im Programm nicht aufgerufen, denn sie stellt das Programm selbst dar. Die Funktion `main()` wird vom Start-Up-Code, der vom Linker dazu gebunden wird, aufgerufen, d.h. das Programm beginnt mit der Ausführung der ersten Anweisung im Funktionskörper von `main()`. Die Funktion main() hat zwei mögliche Formen, mit oder ohne Parameter:

```
int main(void)
{ Körper von main() }
```

oder

```
int main(int argc, char *argv[])
{ Körper von main() }
```

Die erste Form verwendet man, wenn das Programm keine Parameter nimmt, die zweite, wenn Parameter auf der Kommandozeile übergeben werden, die dann im Programm ausgewertet werden sollen.

Im zweiten Fall – `argc` (argument count) und `argv` (argument vector) sind hierbei lediglich traditionelle Namen – bedeutet der erste Parameter die Anzahl der Argumente, incl. des Programmnamens selbst, und ist daher immer mindestens 1. Der zweite Parameter, also `argv`, ist ein Zeiger auf ein Array von Zeigern auf nullterminierte C-Strings, die die Kommandozeilenparameter darstellen. Dieses Array ist selbst auch nullterminiert, also ist `argv[argc]==0`, der sog. Nullzeiger. Der erste Parameter, `argv[0]`, zeigt traditionell auf den Namen, unter dem das Programm aufgerufen wurde. Falls dieser nicht zur Verfügung steht, zeigt `argv[0]` auf den Leerstring, d.h. `argv[0][0]` ist '\0'. Die in `argc` und `argv` gespeicherten Werte, sowie der Inhalt der damit designierten Zeichenketten können vom Programm gelesen und dürfen, wenn gewünscht, auch verändert werden.

Vor Beginn der Ausführung von `main()` sorgt das System dafür, das alle statischen Objekte ihre im Programm vorgesehenen Werte enthalten. Ferner werden zur Interaktion mit der Umgebung drei Dateien geöffnet:

- `stdin` standard input Standardeingabestrom (meist Tastatur)
- `stdout` standard output Standardausgabestrom (meist Bildschirm)
- `stderr` standard error Standardfehlerstrom (meist Bildschirm)

Diese Standardkanäle haben den Datentyp `FILE*` und sind definiert im Header `stdio.h`. In beiden möglichen Formen ist `main()` als `int`-Funktion spezifiziert, der Wert ist die Rückgabe an das aufrufende System und bedeutet den Exit-Status des Programms, der dann z.B. Erfolg oder Misserfolg ausdrücken oder anderweitig vom aufrufenden System ausgewertet werden kann.

Das Programm, bzw. `main()`, endet mit der Ausführung einer return-Anweisung mit einem entspr. Ausdruck, dies ist ein Zuweisungskontext, und es wird in den Typ von `main()`, d.h. nach `int` konvertiert. `main()` endet auch, wenn irgendwo im Programm, d.h. auch innerhalb einer ganz anderen Funktion, die Funktion `exit()`, definiert in `stdlib.h`, mit einem entspr. Wert aufgerufen wird. Dieser Wert gilt dann als Rückgabewert von `main()`.

Wenn `main()` mit einer ausdruckslosen `return`-Anweisung oder an der schließenden Klammer seines Funktionsblocks endet, ist der Rückgabewert unbestimmt. Bei der Beendigung von `main()` werden erst alle mit `atexit()`, ebenfalls definiert in `stdlib.h`, registrierten Funktionen in umgekehrter Reihenfolge ihrer Registrierung aufgerufen. Sodann werden alle geöffneten Dateien geschlossen, alle

mit `tmpfile()` (definiert in `stdio.h`) erzeugten temporären Dateien entfernt und schließlich die Kontrolle an den Aufrufer zurückgegeben.

Das Programm kann auch durch ein – von ihm selbst mit der Funktion `raise()` (definiert in `signal.h`), durch einen Laufzeitfehler (z.B. unbeabsichtigte Division durch Null, illegale Speicherreferenz durch fehlerhaften Zeiger, etc.) oder sonst fremderzeugtes – Signal oder durch den Aufruf der Funktion `abort()` (definiert in `stdlib.h`) terminieren. Was dann im Einzelnen geschieht, wie und ob geöffnete Dateien geschlossen werden, ob temporäre Dateien entfernt werden und was dann der Rückgabestatus des Programms ist, ist implementationsabhängig.

6.2.9 Vektoren und Zeiger

Vektoren (meist *Arrays*, deutsch zuweilen auch Felder genannt) sind als Aggregate komplexe Datentypen, die aus einer Anreihung gleicher Elemente bestehen. Diese Elemente werden aufeinander folgend in Richtung aufsteigender Adressen im Speicher abgelegt, sie können einfache oder selbst auch wieder komplexe Datentypen darstellen. Die Adresse des Arrays ist identisch mit der Adresse des Elements mit der Nummer 0, denn in C werden die Elemente beginnend mit 0 durchnumeriert.

Ein Array wird deklariert mit dem Operator `[]`, in dem die Dimensionsangabe, d.h. die Anzahl der Elemente steht. Die angegebene Anzahl muss eine vorzeichenlose integrale Konstante sein, eine Anzahl 0 ist nicht erlaubt. Der Bezeichner für ein Array ist fest mit seinem Typ verbunden, stellt aber kein Objekt im Speicher dar.

In Zusammenhang mit dem `sizeof`-Operator wird die Größe des Arrays als Anzahl von Einheiten des Typs char geliefert. Der Compiler sorgt für eine korrekte Ausrichtung im Speicher. Ein Beispiel:

```
int iv[10]; /* Ein Array iv von zehn Elementen vom Typ int */
```

Mehr Dimensionen sind möglich, im Gegensatz zu einigen anderen Programmiersprachen gibt es jedoch keine echten mehrdimensionalen Arrays sondern nur Arrays von Arrays (mit beliebiger – vielleicht von der Implementation oder verfügbarem Speicherplatz begrenzter – Anzahl der Dimensionen). Ein Beispiel für die Deklaration eines zweidimensionalen Arrays:

```
double dvv[5][20]; /* Array von 5 Arrays von je 20 doubles */
```

Die Ablage im Speicher erfolgt hierbei zeilenweise (bei zwei Dimensionen), d.h. der der rechte Index (der Spaltenindex bei zwei Dimensionen) variiert am schnellssten, wenn die Elemente gemäß ihrer Reihenfolge im Speicher angesprochen werden. Auf die Elemente zugegriffen wird mit dem Operator `[]`, in dem nun der Index steht, für ein Array mit n Elementen reicht der erlaubte Indexbereich von 0 bis *n*-1.

Beispiel:

```
abc = iv[3]; /* Zuweisung des 4. Elements von iv an abc */
xyz = dvv[i][j]; /* Zuweisung aus dvv, i und j sind
                    Laufvariablen */
```

Die Schrittweite des Index ist so bemessen, dass immer das jeweils nächste – oder vorherige – Element erfasst wird. Es ist erlaubt, negative Indizes oder solche größer als *n*-1 zu verwenden, was jedoch beim Zugriff außerhalb des erlaubten Bereichs des so indizierten Arrays passiert, ist implementationsabhängig (und nicht zu empfehlen!).

Zeiger (*pointer*) sind komplexe Datentypen. Sie beinhalten sowohl die Adresse des Typs, auf den sie zeigen, als auch die Eigenschaften eben dieses Typs, insbesondere, wichtig für die mit ihnen verwendete Adressarithmetik, seine Speichergröße. Zeiger werden deklariert mittels des Operators `*`.

```
int *ip; /* ip ist ein Zeiger auf Typ int */
```

Zeiger müssen initialisiert werden, bevor sie zum Zugriff auf die mit ihnen bezeigten Objekte benutzt werden können:

```
ip = &abc; /* ip wird auf die Adresse von abc gesetzt */
```

Zeiger können nur mit Zeigern gleichen Typs initialisiert werden, oder mit Zeigern auf `void`, (also auf nichts bestimmtes). Zum Initialisieren von Zeigern wird meist der Adressoperator & verwendet, der einen Zeiger auf seinen Operanden erzeugt. In einem Zuweisungszusammenhang gilt der Name eines Arrays als Zeiger auf das erste Element (das mit dem Index 0) des Arrays, d.h. wenn wie im Beispiel weiter oben iv ein Array vom Typ `int` ist:

```
ip = iv; /* gleichwertig mit ip = &iv[0] */
```

Der Zugriff auf das vom Zeiger referenzierte Objekt, (die sog. Dereferenzierung), geschieht mittels des Operators *:

```
if(*ip) /* Test des Inhalts der Var., auf die ip zeigt */
```

Wenn ip auf iv zeigt, dann ist *ip identisch mit iv[0], man hätte auch schreiben können ip[0] oder *iv. Hier zeigt sich nun der grundlegende Zusammenhang zwischen Array- und Zeigernotation in der Sprache C, es gilt:

`a[n]` ist identisch mit `*(a+n)`

Zu beachten ist hierbei lediglich, dass Arraynamen in einem Zugriffskontext feste Zeiger sind (Adressen), sie stellen kein Objekt im Speicher dar und können somit auch nicht verändert werden, wohingegen Zeigervariablen Objekte sind. Ein Zeiger kann inkrementiert und dekrementiert werden, d.h. integrale Größen können addiert oder subtrahiert werden, der Zeiger zeigt dann auf ein dem Vielfachen seiner Schrittweite entsprechend entferntes Objekt.

Zeiger gleichen Typs dürfen miteinander verglichen oder voneinander subtrahiert werden. Wenn sie in den gleichen Bereich (z.B. ein entspr. deklariertes Array) zeigen, ergibt sich eine integrale Größe, die den Indexabstand der so bezeigten Elemente bedeutet. Wenn das nicht der Fall ist, ist diese Operation nicht sinnvoll. Erlaubt (und häufig angewandt) ist auch das Testen des Wertes eines Zeigers.

Die Zuweisung integraler Werte an einen Zeiger hat die Bedeutung einer Adresse des vom Zeiger bezeigten Typs. Wenn die Bedingungen der Ausrichtung dieses Typs (z.B. ganzzahlige Vielfache einer best. Größe) nicht erfüllt sind oder der Zu-

griff auf diese Adresse in der entspr. Typgröße nicht erlaubt sein sollte, kann dies zu Laufzeitfehlern führen. Der Wert 0 eines Zeigers hat die Bedeutung, dass dieser Zeiger ungültig ist, ein sog. Nullzeiger (*null pointer*) – der Zugriff auf die Adresse 0 ist in einem C-System, gleich ob lesend oder schreibend, allgemein nicht gestattet.

Zeiger auf den Typ `void` (also auf nichts bestimmtes) dienen als generische Zeiger lediglich zur Zwischenspeicherung von Zeigern auf Objekte bestimmten Typs. Man kann sonst nichts sinnvolles mit ihnen anfangen, auch keine Adressberechnungen. Sie dürfen ohne weiteres allen Typen von Zeigern zugewiesen werden und umgekehrt.

Initialisierung von Arrays

Wenn erwünscht, können Arrays durch Angabe einer Initialisierungsliste mit konstanten Werten initialisiert werden, hierbei darf dann die Dimensionsangabe fehlen, man spricht dann von einem unvollständigen Arraytyp (*incomplete array type*), und der Compiler errechnet sie selbsttätig aus der Anzahl der angegebenen Elemente der Liste (und komplettiert damit den Typ!):

```
int magic[] = {4711, 815, 7, 42, 3}; /* magic hat 5 Elem. */
```

Ist die Dimension angegeben, werden die Elemente beginnend mit dem Index 0 mit den Werten aus der Liste initialisiert und der Rest, so vorhanden, wird auf 0 gesetzt:

```
long prim[100] = {2, 3, 5, 7, 11}; /* ab Index 5 alles 0 */
```

Die Dimensionsangabe darf nicht geringer als die Anzahl der Elemente der Initialisierungsliste sein:

```
float spec[2] = {1.414, 1.618, 2.718}; /* Fehler! */
```

Die Initialisierung geht auch bei mehr als einer Dimension, hier darf nur die höchste (linke) Dimension fehlen, der Compiler errechnet sie dann:

```
int num[][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}}; /* 3 * 3 */
```

Sind alle Dimensionen angegeben und sind weniger Initialisierer da, werden die restlichen Elemente wie gehabt mit 0 initialiert:

```
int num[3][3] = {{1, 2, 3}, {4, 5, 6}}; /* 3 * 3 */
```

Hier – oder im obigen Beispiel – hätte man die inneren geschweiften Klammern auch weglassen können, denn der Compiler füllt bei jeder Dimension beginnend mit dem Index 0 auf, wobei der rechte Index am schnellsten variiert. Oft besteht jedoch die Gefahr der Mehrdeutigkeit und hilfreiche Compiler warnen hier!

Bei der Initialisierung von char-Arrays mit konstanten Zeichenketten darf man die geschweiften Klammern weglassen:

```
char mword[] = "Abrakadabra"; /* mword hat 12 Elemente */
```

anstatt:

```
char mword[] =
     {'A','b','r','a','k','a','d','a','b','r','a','\0'};
```

oder:

```
char mword[] = {"Abrakadabra"};
```

Auch hier zählt der Compiler wieder die Anzahl der Elemente ab (incl. der terminierenden Null) und dimensioniert das Array selbsttätig. Eine evtl. vorhandene Dimensionsangabe muss mindestens der erforderlichen Anzahl entsprechen, überzählige Elemente werden auch hier mit 0 aufgefüllt:

```
char name[64] = "Heiner Mueller"; /* Ab Index 14 alles 0 */
```

Man beachte folgenden wichtigen Unterschied:

```
char xword[] = "Hokuspokus"; /* xword hat 11 Elemente */
char *xptr = "Hokuspokus"; /* xptr zeigt auf
                              Zeichenkettenkonstante */
```

Im ersten Fall handelt es sich um ein Array namens xword von 11 Elementen in Form eines C-Strings (mit terminierender Null), im zweiten Fall haben wir mit xptr einen Zeiger, der auf einen an anderer Stelle (möglicherweise im Nur-Lesebereich) gespeicherten C-String (jetzt als namenloses Array vom Typ `char`) zeigt.

6.2.10 Strukturen

Eine Struktur (in anderen Sprachen oft als *record*, Verbund, Datensatz bezeichnet) ist als Aggregat ein komplexer Datentyp, der aus einer Anreihung von einer oder mehreren Komponenten (*members*) oft auch verschiedenen Typs besteht, um diese so zusammengefassten Daten dann als Einheit behandeln zu können.

Eine Struktur wird definiert mit dem Schlüsselwort struct gefolgt von einem Block mit den Deklarationen der Komponenten. Beispiel:

```
struct person {
        int num;
        char name[64];
        char email[64];
        char telefon[32];
        char level;
};
```

Hier werden mit dem Schlüsselwort `struct` und dem Bezeichner person, dem sog. Etikett (structure tag), zusammengehörige Daten in einer Struktur zusammengefasst: Es wird ein neuer, benutzerdefinierter Datentyp namens `struct person` geschaffen.

Die Namen der in dem Strukturblock deklarierten Komponenten befinden sich in einem eigenen Namensraum und können nicht mit anderen (äußeren) Namen oder Namen von Komponenten in anderen Strukturen kollidieren. Es wird hierbei auch noch kein Speicherplatz reserviert, sondern lediglich der Typ bekannt gemacht, seine Form beschrieben, also ein Bauplan zur Beschaffenheit dieses Typs und seiner Struktur vorgelegt.

Speicherplatz kann reserviert und somit Variablen dieses Typs erzeugt werden, indem man zwischen der beendenden geschweiften Klammer des Strukturblocks

und dem abschließenden Semikolon eine Liste von Variablennamen einfügt. Übersichtlicher ist wohl aber meist, die Beschreibung der Form von der Speicherplatzreservierung zu trennen. Variablen dieses Typs werden dann z.B. so vereinbart:

```
struct person hugo, *pp; /* 1 Variable und ein Zeiger */
```

Man kann natürlich auch gleich ganze Arrays von diesem neuen Typ erzeugen:

```
struct person ap[100]; /* Array von 100 struct person */
```

Der Compiler sorgt dafür, dass die Komponenten der Strukturen in der Reihenfolge ihrer Deklaration mit der korrekten Ausrichtung angelegt werden und dass die Gesamtheit der Struktur so gestaltet ist, dass sich mehrere davon als Elemente eines Arrays anreihen lassen. Je nach Gestalt der Struktur, abhängig von Maschinenarchitektur und Compiler können dabei zwischen den Komponenten und am Ende der Struktur auch Lücken entstehen, so dass die Gesamtgröße einer Struktur (zu ermitteln mithilfe des `sizeof`-Operators) u.U. größer ist als die Summe der Größen ihrer Komponenten. Der Speicherinhalt der so entstandenen Lücken bleibt dabei undefiniert.

Auf die Komponenten zugegriffen wird direkt mit dem `.`-Operator:

```
hugo.num = 4711; /* Schreibzugriff auf Komp. num von hugo */
```

Der indirekte Zugriff (über Zeiger) geschieht mithilfe des `->`-Operators:

```
pp = &hugo;
pp->level = 12; /* Zugriff auf Komponente level von hugo */
```

Oder entsprechend bei Zugriff auf ein Element eines Arrays:

```
ap[5].num = 4712; printf( "%d", (ap+5)->num );
```

Strukturen können selbst auch wieder (andere) Strukturen als Komponenten enthalten. Erlaubt ist auch die Definition von Strukturen innerhalb des Strukturdefinitionsblocks – dieser Typ ist dann allerdings auch im Sichtbarkeitsbereich der einbettenden Struktur bekannt, daher sollte dies besser vermieden werden. Wenn die Definition des Strukturblocks nicht erfolgt oder noch nicht abgeschlossen ist, spricht man von einem unvollständigen (*incomplete*) Datentyp. Davon lassen sich dann zwar keine Variablen erzeugen – Speicherplatzverbrauch und Gestalt sind ja noch unbekannt, es lassen sich aber schon Zeiger auf diesen Typ erstellen. Auf diese Weise können Strukturen Zeiger auf ihren eigenen Typ enthalten, eine Konstruktion, die oft zur Erzeugung von verketteten Listen verwandt wird. Beispiel:

```
struct mlist {
      struct mlist *prev;
      struct mlist *next;
      char descr[64];
};
```

Strukturen können (an Variablen gleichen Typs) zugewiesen werden, als Argumente an Funktionen übergeben und als Rückgabetyp von Funktionen deklariert werden. Die Zuweisung ist dabei als komponentenweise Kopie definiert. Bei größeren Strukturen empfiehlt sich bei den beiden letzteren Aktionen allerdings, lieber mit Zeigern zu arbeiten, da sonst intern immer über temporäre Kopien

gearbeitet wird, was sowohl zeit- wie speicherplatzaufwendig wäre. Strukturvariablen lassen sich ähnlich wie Arrays mit Initialisierungslisten initialisieren.

Syntaktisch ähnlich einer Struktur ist die Variante oder Union (`union`), mit dem Unterschied, dass die verschiedenen Komponenten nicht nacheinander angeordnet sind, sondern alle an der gleichen Adresse liegend abgebildet werden. Vereinbart werden sie mit dem Schlüsselwort `union`, gefolgt von einem optionalen Etikett, gefolgt von einem Definitionsblock mit den Definitionen der Komponenten, gefolgt von einem Semikolon. Sie werden benutzt, um Daten unterschiedlichen Typs am gleichen Speicherplatz unterbringen zu können (natürlich immer nur einen Typ zur gleichen Zeit!), oder um den Speicherplatz anders zu interpretieren

Der Compiler sorgt dafür, dass die Größe der Union, ihre Ausrichtung inklusive etwaiger Auffüllung den Anforderungen der Maschine entsprechen, daher ist die Größe einer Unionsvariablen immer mindestens so groß wie die Größe ihrer größten Komponente.

Bitfelder

Als mögliche Komponenten von `struct` oder `union` können Bitfelder vereinbart werden. Ein Bitfeld dient zur Zusammenfassung von Information auf kleinstem Raum (nur erlaubt innerhalb `struct` oder `union`). Es gibt drei Formen von Bitfeldern:

- normale Bitfelder (plain bitfields) – deklariert als `int`
- vorzeichenbehaftete (signed bitfields) – deklariert als `signed int`
- nicht vorzeichenbehaftete (unsigned bitfields) – deklariert als `unsigned int`

Ein Bitfeld belegt eine gewisse, aufeinander folgende Anzahl von Bit in einem Integer. Es ist nicht möglich, eine größere Anzahl von Bit zu vereinbaren, als in der Speichergröße des Typs int Platz haben. Es darf auch unbenannte Bitfelder geben, auf die man dann natürlich nicht zugreifen kann, dies dient meist der Abbildung der Belegung bestimmter Register oder Ports. Hier die Syntax:

```
struct sreg {
      unsigned int
      cf:1, of:1, zf:1, nf:1, ef:1, :3,
      im:3, :2, sb:1, :1, tb:1;
};
```

Nach dem Doppelpunkt steht die Anzahl der Bit, die das Feld belegt. Wie der Compiler die Bitfelder anlegt, wie er sie ausrichtet und wie groß er die sie enthaltenden Integraltypen macht, ist völlig implementationsabhängig. Wenn man sie überhaupt je verwenden will, wird empfohlen, sie jedenfalls als `unsigned int` zu deklarieren.

6.2.11 Aufzählungstypen

Aufzählungstypen – Schlüsselwort `enum` – sind benannte Ganzzahlkonstanten (enumeration constants), deren Vereinbarungssyntax der von Strukturen ähnelt. Im Gegensatz zu mit `#define` vereinbarten Konstanten, die der C-Präprozessor (→ 6.3) verarbeitet, werden die enum-Konstanten vom C-Compiler selbst bearbeitet.

Sie sind kompatibel zum Typ, den der Compiler dafür wählt – einen Typ, aufwärtskompatibel zum Typ `int`: Es könnte also auch `char` oder `short` sein, aber nicht `long`, das ist implementationsabhängig – und lassen sich ohne weiteres in diesen überführen und umgekehrt, ohne dass der Compiler prüft, ob der Wert auch im passenden Bereich liegt. Hier einige Beispiele zur Deklaration, bzw. Definition:

```
enum color {red, green, blue} mycolor, hercolor;
enum month {JAN=1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP,
            OCT, NOV, DEC};
enum month mymonth;
enum range {VLO=-10, LLO=-5, LO=-2, ZERO=0, HI=2, LHI=5,
            VHI=10, OVL};
enum range myrange, hisrange;
enum level {AF=-3, BF, CF, DF, EF, FF, GF, HF} xx, yy, zz;
```

Bei aller semantischen Nähe zum Typ `int` sind `enum`-Konstanten oft der beste Weg, um mittels benannter Konstanten das Programm übersichtlicher zu machen und „magische" Zahlen (magic numbers) zu vermeiden, besser oft als die übliche Methode der `#define`-Makros und daher für diesen Zweck sehr zu empfehlen. Diese Art der Verwendung funktioniert natürlich nur für Ganzzahlkonstanten, die den Wertebereich eines `int` nicht überschreiten.

6.2.12 Typdefinitionen

Das Schlüsselwort ist `typedef`. Der Name lässt es zwar vermuten, aber `typedef` dient nicht zur Definition neuer Datentypen, er erzeugt syntaktisch nur andere Namen (Synonyme, Aliasse) für schon bekannte Typen. Das kann, richtig angewandt, zur erhöhten Lesbarkeit des Quelltextes genutzt werden. Einerseits wird `typedef` dazu benutzt, komplizierte oder umständliche Deklarationen zu vereinfachen, andererseits kann durch geschickten Einsatz die Portabilität von Programmcode auf unterschiedliche Umgebungen erhöht werden. Der so erzeugte „neue" Typ ist mit seinem Ursprungstyp voll kompatibel und syntaktisch quasi-identisch. Die Syntax ist: typedef bekannter-Typ neuer-Typname ;. Ein Beispiel:

```
typedef int         int32
typedef short       int16
typedef signed char int8
```

6.3 Der C-Präprozessor

Dem C-Präprozessor obliegt die Vorverarbeitung des C-Quelltexts zur sog. Übersetzungseinheit (*translation unit*), die dann dem eigentlichen Compiler zur Weiterverarbeitung übergeben wird. Er arbeitet als zeilenorientierte Textersetzungsmaschine und versteht die C-Syntax *nicht*.

Seine Aufgabe ist es, jede Zeile mit einem *newline character* abzuschließen, unabhängig von der äußeren Form einer Textzeile, durch ihre Entsprechungen zu ersetzen, Zeilen, die mit einem Rückschrägstrich enden, zusammenzufügen, Zeichengruppen zu ersetzen (z.B. Escape-Sequenzen, Makros), Leerraum zu kondensieren, Kommentare (/* ... */) zu entfernen und durch ein Leerzeichen zu ersetzen, Direktiven auszuführen (auch wiederholt und rekursiv), und Dateien einzufügen (mit denen er dann rekursiv das gleiche anstellt).

Präprozessordirektiven werden mit # eingeleitet. Sie beginnen traditionell am linken Rand und stehen auf einer logischen Zeile. Es gibt folgende Direktiven:

#include <datei.h>	Standard-Header hier einfügen
#include "datei.h"	eigenen Header hier einfügen
#define DIES jenes 17	überall >DIES< durch >jenes 17< ersetzen, sog. Makro
#undef XXX	Makrodefinition XXX entfernen
#line 47	nächste Zeilennummer in der Datei
#error "some failure!"	zur Compilierzeit Fehlermeldung erzeugen
#pragma builtin(xyz)	implementationsdefinierte Option
#ifdef FEATURE	bedingte Compilierung
#ifndef FEATURE	bedingte Compilierung
#if	bedingte Compilierung
#elif	bedingte Compilierung
#else	bedingte Compilierung
#endif	bedingte Compilierung
defined	optional zur Verwendung mit #if und #elif

Präprozessor-Makronamen, wie oben z.B. DIES, XXX und FEATURE, werden traditionsgemäß meist komplett in Großbuchstaben geschrieben, führende Unterstriche sind für das System reserviert und sollten nicht verwendet werden.

Der Präprozessor ist eine reine Textersetzungsmaschine, ohne jegliche Kenntnis von C! Semantische Klarheit von Quellcode hat heutzutage jedoch höchste Priorität. Daher ist der moderne Trend in der Anwendungsentwicklung (ca. seit Beginn der 90er Jahre), den Präprozessor nur noch für einfache Makros sowie Inklusion und – falls notwendig – bedingte Compilierung einzusetzen (siehe hierzu auch Codierungsregel 8, → 6.6). In der Systemsoftware sieht es allerdings etwas anders

aus, wie man leicht beim Studium der Headerdateien feststellen kann. Man traut den Systemprogrammierern offenbar mehr Durchblick und Disziplin zu!

6.4 Die Standardbibliothek

Während es bei früheren Programmiersprachen allgemein üblich war, die Bedienung der Peripherie, Ein- und Ausgabebefehle, Formatieranweisungen für den Druck, spezielle, für den prospektiven Anwendungsbereich erforderliche mathematische oder textverarbeitende Funktionen und ähnliches alles in der Sprache selbst unterzubringen, wurde C von Anfang an ausgelegt, einen möglichst kleinen Sprachkern in Verbindung mit einer Standardbibliothek zu verwenden. Die Sprache sollte es dem Benutzer auf einfache Weise ermöglichen, diese Bibliothek seinem Bedarf anzupassen und auf Wunsch beliebig zu erweitern. Diese Entwurfsphilosophie ist eines der Hauptkennzeichen von C geblieben.

Zur Sprache C gehört eine Standardbibliothek (*standard C library*), deren Programmierschnittstelle (*application programmer interface*, API) über die weiter unten aufgelisteten, insgesamt 18 sog. Header-Dateien definiert wird.

Sie enthalten die Definitionen für Makros und Datentypen, sowie die Deklarationen von Namen und Funktionen in den entspr. Abschnitten der Bibliothek.

<assert.h> <ctype.h> <errno.h> <float.h> <limits.h>

<locale.h> <math.h> <setjmp.h> <signal.h> <stdarg.h>

<stddef.h> <stdio.h> <stdlib.h> <string.h> <time.h>

<iso646.h> <wchar.h> <wctype.h>

Die Anbindung entspr. Abschnitte der Standardbibliothek sollte im Quelltext immer über die Einbindung der jeweils zutreffenden Header mittels der #include <...> Präprozessoranweisungen (→ 6.3) geschehen, um die notwendigen Definitionen alle korrekt zu übernehmen.

Die Einbindung der entspr. Teile der Objektbibliothek durch den Linker geschieht meist automatisch, zuweilen ist es jedoch notwendig, bestimmte Teile – oft die zu math.h gehörenden Funktionen – mittels spezieller Linkeroptionen bei der Compilierung explizit anzufordern.

6.5 Wie arbeitet ein C-Compiler?

Im letzten Teil dieses Kapitels zur Einführung in die imperative Sprache C wird noch ein kurzer Blick auf die Übersetzung von C in Maschinensprache geworfen. Dies erfolgt aus einem besonderen Grund, denn anhand des so genannten Zwischencodes kann man schon vergleichsweise gut auf die in Abschnitt 3.2.3 eingeführten Worst-Case-Execution-Times (WCET) schließen. Doch zunächst folgt einmal ein Blick auf die Compilerphasen.

6.5.1 Compilerphasen

Die Übersetzung eines in C geschriebenen Programms erfolgt in insgesamt 4 Phasen, von denen der Compiler an zweien unmittelbar beteiligt ist. Die 4 Phasen sind:

- Präprozessorphase
- Frontendphase des Compilers
- Backendphase des Compilers
- Linkerphase

Die Präprozessorphase wurde bereits in Abschnitt 5.3 erwähnt, hierbei handelt es sich um eine Vorbereitung des zu übersetzenden Sourcecodes. Textmakros werden ersetzt, die so genannten include-Dateien eingesetzt, Kommentare gelöscht, die Zeilen immer durch ein Newline-Zeichen getrennt usw. Der Output dieser Phase ist ein reiner Sourcecode, der bislang noch keine Überprüfung oder Übersetzung erfahren hat.

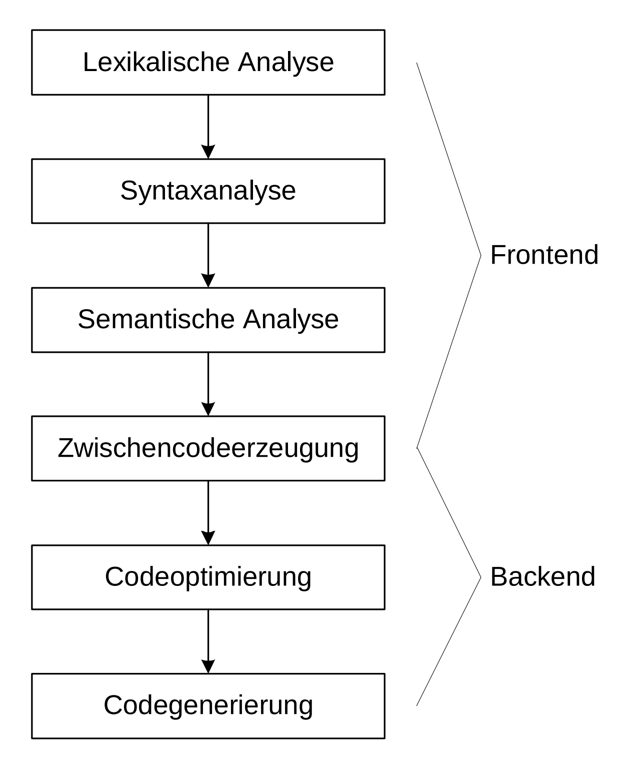


Bild 6.1 Die Phasen eines Compilers [ASU99]

Im Frontend des Compilers (→ Bild 6.1) wird dieser Sourcecode eingelesen (Scanner) und überprüft (Parser). Ziel ist es dabei, die korrekte Syntax zu überprüfen, eine erste Syntaxumwandlung und erste Optimierungen durchzuführen. Das Ziel dieser Phase ist ein Zwischencode, der noch von dem Zielsystem (dem Mikroprozessor) unabhängig ist, aber dennoch die Umsetzung in Assembler- oder Maschinensprache vorbereitet. Der Output dieses Frontendteils wird im nächsten Abschnitt genauer betrachtet.

Im Backend des Compilers erfolgt das Einlesen des Zwischencodes (*intermediate representation, IR*), die Umsetzung in Assemblersprache einschließlich der maschinenspezifischen Optimierung und der Assemblerlauf. Ziel dieses Abschnitts ist der so genannte Objektcode, der neben dem Maschinencode – noch unvollständig – auch Informationen zu den Daten und Programmabschnitten mitführt.

Der Linker liest dann abschließend den Objektcode ein, dazu die angegebenen Standard- und spezifischen Bibliotheken, und fügt das zusammen. Nunmehr sind alle Adressen, auch die der aus der Bibliothek genutzten Funktionen (wie etwa printf) bekannt, und der Maschinencode kann mit allen Adressen vervollständigt werden. Output des Linkers ist ein ausführbarer Maschinencode (in einem Fileformat).

6.5.2 Die Erzeugung des Zwischencodes [Sie07a]

Für den Zwischencode existiert kein genormtes Format, jeder Compiler nutzt dort seine hauseigene Syntax. Besonders interessant ist jedoch das Lance2-Compilersystem [Lance2], das aus C ein low-level-C erzeugt und dieses als Zwischencode nutzt. Diese Untermenge von C, die dieses Compilersystem als Zwischencode (Intermediate Representation, IR) nutzt, ist natürlich beschränkt. Wesentliche Merkmale sind:

Anweisungen (Statements):

- Zuweisungen (Assignments): a = b + c, y = function(a, b), ...
- Sprünge (Jumps): goto label_1; (diese Sprünge sind Compiler-berechnet und somit ”zugelassen“)
- Bedingte Sprünge (Conditional Jumps): if(cond) goto label_2;
- Marken (Label): label_1:
- Rücksprung ohne Rückgabewert (return void): return;
- Rücksprung mit Rückgabewert (return value): return x;

Ausdrücke (Expressions):

- Symbole: main, a, count …
- Binäre Ausdrücke (binary expressions): a * b, x / y …
- Unäre Ausdrücke (unary expressions): ~a, *p …
- Type Casts: (int), (char)

- Konstanten (in verschiedenen Formaten): -5, 3.141592653589

Ein kurzer Blick in die obige Liste verrät, dass bei den Anweisungen Schleifen wie for, while und do .. while komplett fehlen. Diese Schleifen werden durch die aufgezählten Konstrukte abgebildet bzw. in diese übersetzt, und es gilt noch zu zeigen, wie dies erfolgt.

Der wichtigste Zusatz, das Zwischencodeformat betreffend, besteht noch in der Beschränkung der Ausdrücke und der Zuweisungen: Sie werden auf ein 3-Adressformat eingeschränkt, d.h., eine Wertzuweisung an ein links stehendes Symbol (a = ...) wird rechtsseitig durch einen unären oder einen binären Ausdruck bestimmt. Längere „Kettenrechnungen“ müssen dementsprechend in Teilrechnungen mit Einfügung temporärer Variablen geteilt werden, eine Aufgabe, die dem Compiler zufällt (zu den Problemen mit Seiteneffekten und Sequenzpunkte siehe hier Anmerkungen in Abschnitt 6.2.4). Der Grund für diese Einschränkung ist sehr offensichtlich: Dem 3-Adressformat entsprechen häufig direkt Assemblerbefehle (etwa: ADD R3, R1, R2, was R3 = R1 + R2 bedeutet).

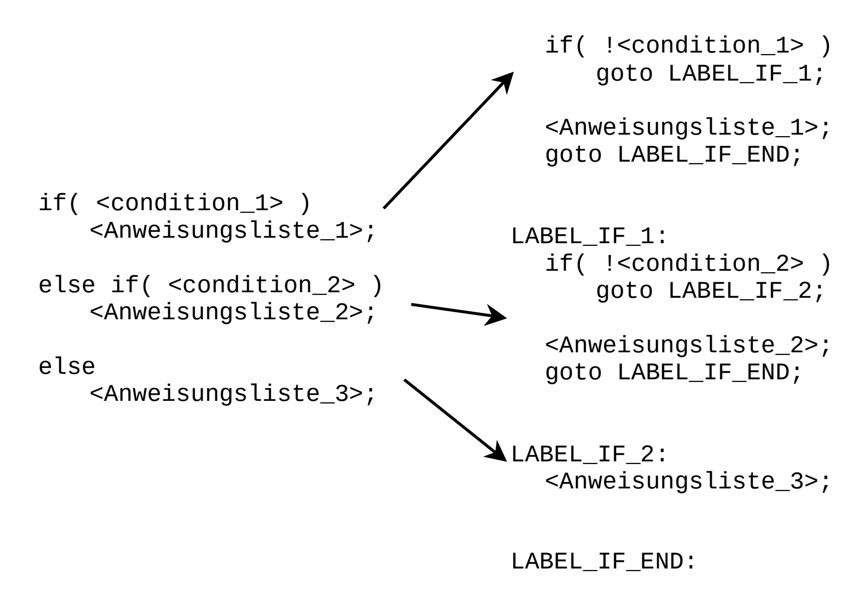


Bild 6.2 Übersetzung der if/else if/else-Verzweigung

Bild 6.2 zeigt die Übersetzung einer if/else if/else-Verzweigung. Dabei wird deutlich, dass nur if-Konstrukte mit anschließendem Sprung (also zusammengefasst der ”bedingte Sprung“) genutzt werden. Die Bedingungen selbst müssen dabei invertiert ausgewertet werden, da ja die Liste der Anweisungen, die bei Erfüllung der ursprünglichen Bedingung auszuführen sind, nun übersprungen werden.

Dies mag etwas holprig wirken, denn bei Zulassung einer üblichen if-Verzweigung wäre dies wesentlich einfacher zu übersetzen. Diese Form der Übersetzung hat jedoch den entscheidenden Vorteil, dass nur bedingte *Sprünge* verwendet werden, und die lassen sich 1:1 in eine Sequenz von Assemblerbefehlen wie etwa

```
cmp R1, R2;           Auswertung der Bedingung
beq LABEL_IF_1;       Bedingter Sprung
```

übersetzen.

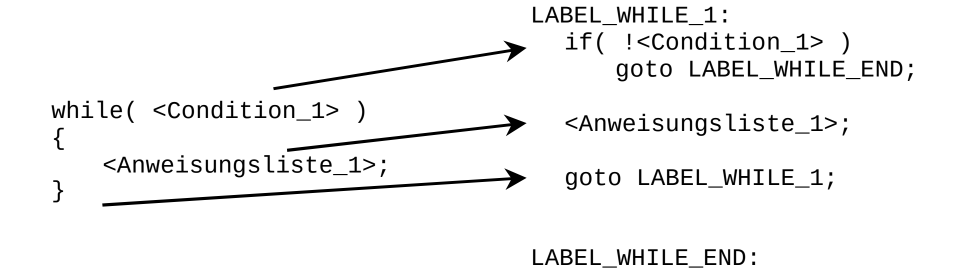


Bild 6.3 Übersetzung einer `while`-Schleife

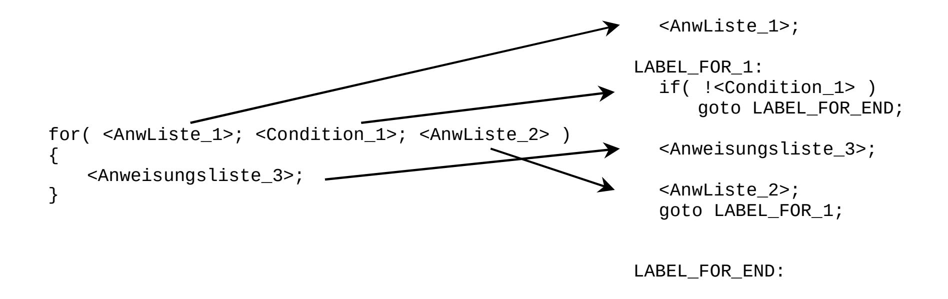


Bild 6.4 Übersetzung einer `for`-Schleife

Bild 6.3 zeigt die Übersetzung der `while`-Schleife, Bild 6.4 die der etwas komplexeren `for`-Schleife. In beiden Fällen werden bedingte und unbedingte Sprünge verwendet, um die Schleifenstruktur entsprechend abzubilden, wobei die Bedingung auch wieder invertiert verwendet werden müssen, um den Sprung aus der Schleife zu beschreiben. Entsprechend den hier gezeigten Codeabschnitten können nun auch die switch/case-Verzweigung (Bild 6.5) und die do..while-Schleife (Bild 6.6) übersetzt werden, wobei die Mehrfach-Fallunterscheidung (switch/case) etwas komplexer ist.

```
switch( <Ausdruck_1> )
{
   case Constant_1:
     <Anweisungsliste_1>;
     break;

   case Constant_2:
     <Anweisungsliste_2>;
     break;

   default:
     <Anweisungsliste_3>;
     break;
}
```

```
  Condition_1 = (<Ausdruck_1> == Constant_1);
  if( !<Condition_1> )
     goto LABEL_CASE_1;

  <Anweisungsliste_1>;
  goto LABEL_CASE_END;

LABEL_CASE_1:
  Condition_2 = (<Ausdruck_1> == Constant_2);
  if( !<condition_2> )
     goto LABEL_CASE_2;

  <Anweisungsliste_2>;
  goto LABEL_CASE_END;

LABEL_CASE_2:
  <Anweisungsliste_3>;

LABEL_CASE_END:
```

Bild 6.5 Übersetzung der switch/case-Verzweigung

```
do
{
   <Anweisungsliste_1>;
}
while( <Condition_1> )
```

```
LABEL_DO_WHILE_1:

  <Anweisungsliste_1>;

  if( <Condition_1> )
     goto LABEL_DO_WHILE_1;
```

Bild 6.6 Übersetzung einer do .. `while`-Schleife

6.5.3 Laplace-Filter als Beispiel [Sie07b]

Ein zweifellos sehr einfaches Filterprogramm zur Bildverarbeitung besteht in dem Laplace-Filter [Laplace], das näherungsweise die zweite Ableitung eines zweidimensionalen Feldes bildet und dadurch die Kantendetektierung ermöglicht.

Der Algorithmus basiert darauf, dass für jeden neu zu berechnenden Punkt in einem Kantenbild der Wert aus dem ursprünglichen Bild für diesen Punkt genommen wird, mit dem Gewichtsfaktor 4 multipliziert wird, und von diesem Wert dann die unmittelbaren Nachbarn (keine Diagonalen) subtrahiert werden. Um die Darstellung des Übersetzungsvorgangs möglichst einfach zu halten, wird hier die eindimensionale Variante gewählt (Bild 6.7).

Dieser Code wird nun entsprechend dem Lance2-Compilers [Sie07a] in einen Zwischencode wie in 6.5.2 beschrieben übersetzt.

```
01:  #define X_DIM 100
02:  short bild[X_DIM];
03:  short kanten[X_DIM];
04:
05:  void laplace_filter_1d(void)
06:  {
07:      short int x;
08:
09:      for( x = 1; x < X_DIM - 1; x++ )
10:      {
11:          kanten[x] = 2 * bild[x] – bild[x-1] - bild[x+1];
12:      }
13:
14:  }
```

Bild 6.7 Sourcecode für eindimensionales Lapace-Filter

Konkret werden im Zwischencode nur wenige Operationen benötigt, so z.B.:

- Wertzuweisungen an Variable, hierbei rechtsseitig einfache Rechnungen mit maximale zwei Operanden und einer Operation; dies wird auch für Adressrechnung bei indizierten Variablen benötigt.
- Vergleiche mit Zuweisung des Booleschen Werts an eine Variable
- `if`-Konstrukt mit einer Variablen, auf deren Wahrheitswert verglichen wird, mit anschließendem (Compiler-berechnetem) `goto`.

```
101: short bild[100];
102: short kanten[100];
```

```
103: void laplace_filter_1d()
104: {
105:  short x_3;
106:  short t1;
107:  short t2;
108:  short t3;
109:  short *t4;
110:  short t5;
111:  short t6;
112:  short t7;
113:  short t8;
114:  short *t9;
115:  short t10;
116:  short t11;
```

```
117:  short *t12;
118:  short t13;
119:  short t14;
120:  short *t15;
121:  short t16;
122:  short t17;
123:  short *t18;
124:  short t19;
125:  short t20;
126:  short *t21;
127:  short t22;
128:  short *t23;
129:  short t24;
```

Bild 6.8 Durch den Frondendteil erzeugter Zwischencode für die Zeilen 1-7 (Bild 6.7)

```
130: x_3 = 1;                    /* Initialisierung for-Schleife */
131: LL1:
132: t1 = x_3 < 99;              /* Berechnung ggf. Schleifenende */
133: t24 = !t1;
134: if( t24 ) goto LL2;

135: t4 = (short * ) bild;       /* Berechnung &(bild[x]) */
136: t5 = x_3 * 2;
137: t6 = t4 + t5;
138: t7 = *t6;                   /* Zugriff auf bild[x] */
139: t8 = t7 * 2;

140: t9 = (short *) bild;
141: t10 = x_3 – 1;              /* Berechnung &(bild[x-1]) */
142: t11 = t10 * 2;
143: t12 = t9 + t11;
144: t13 = *t12;                 /* Zugriff auf bild[x-1] */

145: t14 = t8 – t13;             /* Berechnung 2 * bild[x] – bild[x-1] */

146: t15 = (short *) bild;       /* Berechnung &(bild[x+1]) */
147: t16 = x_3 + 1;
148: t17 = t16 * 2;
149: t18 = t15 + t17;
150: t19 = *t18;                 /* Zugriff auf bild[x+1] */

151: t20 = t14 – t19;            /* 2 * bild[x] – bild[x-1] – bild[x+1] */

152: t21 = (short *) kanten;     /* Berechnung &(kanten[x]) */
153: t22 = x_3 * 2;
154: t23 = t21 + t22;
155: *t23 = t20;                 /* kanten[x] = ... */

156: t2 = x_3;                   /* Inkrement der Variablen x */
157: t3 = t2 + 1;
158: x_3 = t3;
159: goto LL1;

160: LL2:                        /* Schleifenende */
161: return;
162: }
```

Bild 6.9 Zwischencode für den eindimensionalen Laplace-Filter

Generierung des Zwischencodes

Diese Form des Zwischencodes bedeutet aber auch, dass voraussichtlich eine Vielzahl von temporären Variablen benötigt wird, da viele Zwischenrechnungen zu

machen sind. Im Zwischencode – hier werden die Zeilen ab 101 durchnummeriert – in Listing in Bild 6.8 fällt sofort die Vielzahl der Variablen auf, die hier zusätzlich zum Originalcode deklariert werden. Die Zeilen 101-102 entsprechen der Deklaration der Arrays in C, sie wird die Größe 100 haben. Die in der Funktion `laplace_filter_1d` (Bild 6.7) deklarierte Variable taucht in Zeile 105 wieder auf, sie werden lediglich zusätzlich mit einem '_' versehen und durchnummeriert, ansonsten entspricht diese Deklaration der von C.

Die nun folgenden Variablen sind alles vom Compiler erzeugte temporäre Variablen. Man erkennt sie gut an dem fehlenden Unterstrich im Namen. Diese Variablen werden für Berechnungen gebraucht, die im C-Code noch ohne Zwischenschritte auskamen, und sind eine Folge der Übersetzung in der 3-Adresscode. Da der Compiler temporäre Variabeln nicht wieder verwendet, legt er erst einmal für jede dieser Berechnungen eine neue temporäre Variable an. Etwaige Optimierungen sind Aufgabe des nachfolgenden Codegenerators.

Bild 6.9 zeigt den entstehenden Zwischencode bei den gegebenen Voraussetzung. Die einzelnen Berechnungen erscheinen recht komplex, etwa für den Zugriff auf bild[x+1] (Zeile 146-150), sie sind aber notwendig, und in jeder Zeile wird das 3-Adressformat eingehalten. Der Zugriff auf `bild[x+1]` bedeutet nicht anderes, als dass die Adresse `&bild[0] + (x+1)*sizeof(bild[0])` gebildet und dann auf den Inhalt lesend oder schreibend zugegriffen wird. Diese Berechnung der Zugriffsadresse ist in den Zeilen 146 (Basisadresse) sowie 147-149 (Indexberechnung) codiert und steht dann in der Variablen t18 zur Verfügung. Di Operation `sizeof(bild[0])` liefert dabei den 2 für short.

Das Setzen der Basisadresse &bild[0] findet offenbar mehrfach in den Zeilen 135, 140 und 146 statt. Dies kann eventuell optimiert werden, wobei der Compiler exakt prüfen muss, ob nicht durch externe Programmteile – eine Interrupt Service Routine etwa – die Adresse verändert werden könnte, da bild[] global definiert wurde. In diesem Fall kann aber die Adresse selbst nicht verändert werden – sie ist konstant, nur die Inhalte sind variabel – so dass die Zeilen 140 und 146 entfallen können (→ nächsten Abschnitt).

Die Zeilen 132 bis 134 stellen die Auswertung der Schleifenendebedingung dar. Zunächst wird der aktuelle Wert von x_3 mit 99 (X_DIM-1) verglichen, und das Vergleichsergebnis wird der Compiler-generierten Variablen t1 zugewiesen. Diese Variable fungiert als Boole'sche Variable, d.h., sie soll nur Werte für true und false speichern. Die Zuweisung des negierten Wahrheitswerts an t24 in Zeile 133 und die Auswertung durch einen bedingten Sprung (Zeile 134) komplettieren diesen Abschnitt.

Vom Zwischencode zum Assemblercode

Die Übersetzung des Zwischencodes in einen Assemblercode soll anhand einer Modell-CPU erfolgen. Diese wird als MPM3, Mikroprozessormodell #3, bezeichnet und stellt einen RISC-Prozessor mit einer intrinsischen Verarbeitungsbreite von

16 bit dar [Sie04]. Dieses Modell wurde gewählt, weil die typischen Vorgänge daran sehr gut gezeigt werden können, ohne auf die Spezialitäten einer marktgängigen Architektur eingehen zu müssen.

Die spontane Übersetzung des Zwischencodes wird durch einen weiteren Vorgang, die Abbildung der Variablen auf Register oder Speicher betreffend, gebremst. Die Abbildung auf den Maschinencode ist tatsächlich nicht besonders schwierig, hingegen sind die Anforderungen an die Daten schon schwieriger erfüllbar. Bei diesem Vorgang müssen insbesondere Randbedingungen wie Laufzeitminimierung erfüllt werden.

```
301:  ORG $0200
302:  BILD      DW 0
303:  ORG $264
304:  KANTEN DW 0
305:  ORG $0300
306:  _laplace_filter_1d:
```

Bild 6.10 Assemblercodegenerierung für die Zwischencodezeilen 101-103

Die Übersetzung des ersten Teils des Zwischencodes – Bild 6.8 – fällt überraschend klein aus. Die gesamte Abbildung der doch eher großen Menge an Variablen erfolgt so, dass lediglich die Variablen bild[] und kanten[] im Speicher angelegt und mit Werten initialisiert wird. Die Initialisierung erfolgt dem Datentyp short gemäß mit 16 bit (DW, define word). Die übrigen Variablen werden auf 7 der vorhandenen 8 Datenregister R0 bis R7 abgebildet (Tabelle 6.4).

Tabelle 6.4 Zuordnung der Variablen zu Registern

Variable	*Register*
x	R0

Variable	*Register*
t4, t9, t15, t21	R1
t5, t22	R2
t6, t23	R3
t7, t8, t14, t15, t20	R4
t10, t11, t12, t16, t17, t18	R5
t13, t19	R6

Es entfallen:
t1, t2, t3, t24

Bild 6.11 zeigt die Übersetzung des Zwischencodes in den Assemblercode und hierbei auch einige Optimierungsmöglichkeiten (die keineswegs immer im Backendgenerator vorhanden sein werden). Wie bereits dargestellt können die Zeilen 140 und 146 durch Optimierung entfallen (im Übrigen bereits im Zwischencode).

```
130: x_3 = 1;              ──────────▶ 307: MOV  R0, #1
131: LL1:                              308: LL1:
132: t1 = x_3 < 99;        ──────────▶ 309: CMP  R0, #99
133: t24 = !t1;
134: if( t24 ) goto LL2;   ──────────▶ 310: BGE  LL2;

135: t4 = (short * ) bild; ──────────▶ 310: MOV  R1, #(bild & #$ff)
                                       311: MOVH R1, #(bild >> 8)
136: t5 = x_3 * 2;         ──────────▶ 312: ASL  R2, R0;
137: t6 = t4 + t5;         ──────────▶ 313: ADD  R3, R1, R2
138: t7 = *t6;             ──────────▶ 314: LD   R4, [R3]
139: t8 = t7 * 2;          ──────────▶ 315: ASL  R4, R4

140: t9 = (short *) bild;
141: t10 = x_3 - 1;        ──────────▶ 316: DEC  R5, R0
142: t11 = t10 * 2;        ──────────▶ 317: ASL  R5, R5
143: t12 = t9 + t11;       ──────────▶ 318: ADD  R5, R1, R5
144: t13 = *t12;           ──────────▶ 319: LD   R6, [R5]

145: t14 = t8 - t13;       ──────────▶ 320: SUB  R4, R4, R6

146: t15 = (short *) bild;
147: t16 = x_3 + 1;        ──────────▶ 321: INC  R5, R0
148: t17 = t16 * 2;        ──────────▶ 322: ASL  R5, R5
149: t18 = t15 + t17;      ──────────▶ 323: ADD  R5, R1, R5
150: t19 = *t18;           ──────────▶ 324: LD   R6, [R5]

151: t20 = t14 - t19;      ──────────▶ 325: SUB  R4, R4, R6

152: t21 = (short *) kanten; ────────▶ 326: MOV  R1, #(kanten & #$ff)
                                       327: MOVH R1, #(kanten >> 8)
153: t22 = x_3 * 2;        ──────────▶ 328: ASL  R2, R0
154: t23 = t21 + t22;      ──────────▶ 329: ADD  R3, R1, R2
155: t23 = t20;            ──────────▶ 330: ST   [R3], R4

156: t2 = x_3;
157: t3 = t2 + 1;
158: x_3 = t3;             ──────────▶ 331: INC  R0, R0
159: goto LL1;             ──────────▶ 332: JMP  LL1

160: LL2:                  ──────────▶ 333: LL2:
161: return;               ──────────▶ 334: RTS
162: }
```

Bild 6.11 Abbildung der Zwischencodezeile 130-162 in Assemblercode.

Die Optimierung kann sogar noch weitergehen: Das Programm läuft ein Weile in der Schleife von LL1 (Zeile 131) bis goto LL1 (Zeile 159), und in der gesamten Schleife werden der Wert für short *bild (in R1) und short *kanten (ebenfalls in R1) drei- bzw. einmal im Register geschrieben und dann lesend benutzt. Wenn man also den Wert für short *kanten in einem anderen Register speichern kann (z.B. in R7), dann könnten die Zeilen 310/311 und 326/327 im Assemblercode nach oben, also zwischen Zeile 307 und 308 geschoben werden. Dieses Verfahren, *Instruction Scheduling* genannt, bewirkt in diesem Fall, dass die Zeilen nur einmal für alle Schleifen durchlaufen wird und spart somit Rechenzeit.

Noch kurz ein Wort zu den vielleicht ungewöhnlich ausschauenden Zeilen 310/311 bzw. 326/327. In diesen Assemblercodezeilen wird ein Register mit einer 16-bit-Konstanten geladen. Die zugrunde liegende Architektur ist (angenommen) ebenfalls mit 16-bit-Datenstrukturen (Register, ALU) und 16-bit Adressen versehen, und hierbei kommt es zum Problem. Bei RISC-Prozessoren ist es sozusagen Pflicht, dass ein Befehl in einem Takt bearbeitet wird, und wenn nun ein Befehl eine Breite von 16 bit im Speicher hat, dann passen keine Konstanten mit 16 bit Breite dort hinein (weil ja die Operation auch noch beschrieben werden muss).

Die Lösung besteht in dem Befehlspaar MOV/MOVH (Move und Move High Byte). Der erste Teil kopiert den Operanden – der untere Teil der Adresse für bild bzw. kanten – in die entsprechenden Bits des Registers, meist mit Belegung auch der oberen Bits (auf 0 oder mit Vorzeichenerweiterung), und MOVH kopiert dann den Operanden, dem oberen Teil der Adresse entsprechend, in die oberen Bits des Registers.

Letztes Augenmerk soll noch auf die Übersetzung der bedingten Sprünge gelegt werden (Zeile 132-134). Der Zwischencode bestand hier aus drei Anweisungen: die Bedingung wird auf ihren Wahrheitswert berechnet, der Wert wird invertiert, und unter Auswertung dieses entstehenden booleschen Wertes wird dann gesprungen (oder nicht). Im Assemblercode treten hiervon nur noch zwei Anweisungen auf: Die Auswertung der Bedingung wird auf das Setzen von Flags abgebildet (Zeile 309), und diese Flags werden – mit logischer Invertierung, denn BGE bedeutet branch if greater or equal, was die umgekehrte Bedingung zu kleiner (less than) ist – dann in Zeile 310 ausgewertet.

6.5.4 Optimierungsmöglichkeiten

Im vorangegangenen Beispiel wurden die Zeilen 140 und 146 wegoptimiert, weil es sich offensichtlich immer um die gleiche Zuweisung handelt. Im Allgemeinen gilt, dass derartige Optimierung der mehrfachen Wertzuweisung durchgeführt werden können, wenn sichergestellt ist, dass Wertänderungen – auch durch externe Routinen wie ISR – ausgeschlossen sind.

Diese Optimierung, die bereits im Zwischencode erfolgen kann, ist immer dann möglich, wenn die entsprechende Variable

- lokal angelegt ist oder
- bei globaler Speicherklasse konstant ist.

Im Fall der Beispiels des eindimensionalen Laplace-Filters (Bild 6.7) galt die zweite Voraussetzung, da die Adresse des Arrays `bild[]`, also `&(bild[0])`, konstant angelegt wird.

6.5.5 Zusammenhang zwischen Zwischencode und WCET

Abschließend soll der Zusammenhang zwischen dem Zwischencode und den Worst-Cae-Execution-Times beleuchtet werden. Das Beispiel aus dem vorigen Abschnitt ist dabei richtungsweisend, denn der (fast) lineare Zusammenhang zwischen Zwischencode und Assemblercode wird immer beobachtet.

Um einen Schätzwert für die WCET zu erhalten, muss man also den Codezeilen des Zwischencodes per Tabelle maximale Ausführungszeiten zuordnen und zusammenzählen. Da der Code nicht optimiert ist, liegt die Vermutung nahe, dass die berechneten Zeiten wirklich maximale Zeiten darstellen.

In der Praxis jedoch gibt es dabei Detailprobleme. Abgesehen davon, dass die oben geäußerte Vermutung kein Beweis ist (in der Praxis aber „nie“ widerlegt wird), sollten die berechneten WCETs aber auch realistisch sein, und hier wird es schwieriger.

Als Beispiel sei die Zeile 137 aus Bild 6.11 betrachtet:

```
137:   t6 = t4 + t5;
```

Diese wird in die Assembleranweisung

```
313:   ADD  R3, R1, R2
```

übersetzt, also ein einfacher Additionsbefehl mit einer Laufzeit von einem Takt, was dann der geschätzten WCET für diese Zeile entspricht. Dieser eine Takt gilt aber nur, wenn die drei temporären variablen t4, t5 und t6 in jeweils einem Register gehalten werden. Sind diese Variablen (aus Registermangel) im Speicher, sieht die Übersetzung für diese Architektur ganz anders aus:

```
313a:  MOV  R7, #(t4 & #&ff)
313b:  MOVH R7, #(t4 >> 8)
313c:  LD   R1, [R7]
313d:  MOV  R7, #(t5 & #&ff)
313e:  MOVH R7, #(t5 >> 8)
313f:  LD   R2, [R7]
313g:  ADD  R3, R1, R2
313h:  MOV  R7, #(t6 & #&ff)
313i:  MOVH R7, #(t6 >> 8)
313j:  ST   [R7], R3
```

Nunmehr sind es 10 Assemblerzeilen geworden, also eine WCET von 10 Takten, weil für jedes Laden erst die Adresse in ein Register, dann der Speicherinhalt geladen werden muss, dann addiert wird, und dann das Ergebnis wieder zurückgeschrieben wird.

Rechnet man also mit einer WCET von 10 für die Zeile 137, kann man diese garantieren, aber die Gefahr, dass nun drastisch überschätzt wird, ist groß. Dieser Widerspruch kann aktuell nur gelöst werden, wenn die WCET auf Maschinencodeebene bestimmt oder geschätzt wird.

6.6 Coding Rules

Abschließend in diesem Kapitel sollen – beispielhaft – Codierungsregeln (Coding Rules) zitiert werden, die gerade für Softwareentwicklung in sicherheitskritischen Bereichen gelten und anerkannt sind. Über Codierungsregeln kann man sich natürlich sehr ausführlich auslassen, jede Firma, jede Entwicklungsgruppe, die etwas auf sich hält, hat mindestens ein Regelwerk, das auch sehr umfänglich sein kann. Die hier zitierten Regeln [Hol06] stellen mit einer Anzahl von 10 ein übersichtliches Regelwerk dar.

Regel 1:

Im gesamten Code sollen nur einfache Kontrollflusskonstrukte verwendet werden. Insbesondere sollen *goto*, direkte oder indirekte Rekursion vermieden werden.

Dies resultiert insbesondere in einer erhöhten Klarheit im Code, der leichter zu analysieren und zu beurteilen ist. Die Vermeidung von Rekursion resultiert in azyklische Codegraphen, die wesentlich einfacher bezüglich Stackgröße und Ausführungszeit analysiert werden können.

Die Regel kann noch dadurch verschärft werden, dass pro Funktion nur ein einziger Rücksprung erlaubt ist.

Regel 2:

Alle Schleifen müssen eine Konstante als obere Grenze haben. Es muss für Codecheck-Tools einfach möglich sein, die Anzahl der durchlaufenen Schleifen anhand einer Obergrenze statisch bestimmen zu können.

Diese Regel dient dazu, unbegrenzte Schleifen zu verhindern. Hierbei müssen auch implizit unbegrenzte Schleifen wie das folgende Beispiel verhindert werden, die wichtige Regel ist also diejenige, dass der Codechecker die Obergrenze erkennen können muss.
Es gibt allerdings eine Ausnahme von dieser Regel: Es gibt immer wieder explizit unendlich oft durchlaufene Schleifen (etwa: `while(1)`), die für bestimmte Aufgaben notwendig sind (Process Scheduler, Rahmen für endlos laufendes Programm etc.). Diese sind selbstverständlich erlaubt.

```
int k, m, array[1024];

for( k = 0, m = 0; k < 10; k++, m++ )
{
        if( 0 == array[m] )
            k = 0;
}
```

Beispiel 5.1 Implizit unbegrenzte `for`-Schleife (als Negativbeispiel)

Eine Möglichkeit, diese Regel zu erfüllen und bei Überschreiten dieser oberen Grenze einen Fehler bzw. eine Fehlerbehebung einzuführen, sind so genannte `assert()`-Funktionen (siehe auch Hardwarebeschreibungssprachen wie VHDL). Bei Überschreiten wird eine solche Funktion aufgerufen, diese kann dann entsprechende Aktionen einleiten. Es ist zwar möglich, die Fehlerbehebung auch in den eigentlichen Sourcecode einzubauen, die explizite Herausführung dient aber der Übersicht.

Regel 3:

Nach einer Initialisierungsphase soll keine dynamische Speicherallokation mehr erfolgen.

Die Allokationsfunktionen wie `malloc()` und die Freigabe (`free()`) sowie die Garbage Collection zeigen oftmals unvorhersagbare Verhaltensweisen, daher sollte hiervon im eigentlichen Betrieb Abstand genommen werden. Zudem stellt die dynamische Speicherverwaltung im Programm eine hervorragende Fehlerquelle dar bezüglich Speichernutzung nach Rückgabe, Speicherbereichsüberschreitung etc.

Regel 4:

Keine Funktion soll mehr als 60 Zeilen haben, d.h. bei einer Zeile pro Statement und pro Deklaration soll die Funktion auf einer Seite ausgedruckt werden können.

Diese Regel dient einfach der Lesbarkeit und der Übersichtlichkeit des Codes.

Regel 5:

Die Dichte an Assertions (siehe auch Regel 2) soll im Durchschnitt mindestens 2 pro Funktion betragen. Hierdurch sollen alle besonderen Situationen, die im Betrieb nicht auftauchen dürfen, abgefangen werden. Die Assertions müssen seiteneffektfrei sein und sollen als Boolesche Tests definiert werden.

Die `assert()`-Funktionen selbst, die bei fehlgeschlagenen Tests aufgerufen werden, müssen die Situation explizit bereinigen und z.B. einen Fehlercode produzieren bzw. zurückgeben.

Untersuchungen zeigen, dass Code mit derartigen Assertions, die z.B. Vor- und Nachbedingungen von Funktionen, Werten, Rückgabewerten usw. testen, sehr defensiv arbeitet und einer raschen Fehlerfindung im Test dient. Die Freiheit von Seiteneffekten lässt es dabei zu, dass der Code bei Performance-kritischen Abschnitten später auskommentiert werden kann.

Regel 6:

Alle Datenobjekten müssen im kleinstmöglichen Gültigkeitsbereich deklariert werden.

Dies ist das Prinzip des Versteckens der Daten, um keine Änderung aus anderen Bereichen zu ermöglichen. Es dient sowohl zur Laufzeit als auch zur Testzeit dazu, den Code möglichst einfach und verständlich zu halten.

Regel 7:

Jede aufrufende Funktion muss den Rückgabewert einer aufgerufenen Funktion checken (falls dieser vorhanden ist), und jede aufgerufene Funktion muss alle Aufrufparameter auf ihren Gültigkeitsbereich testen.

Diese Regel gehört wahrscheinlich zu den am meisten verletzten Regeln, aber der Test z.B. darauf, ob die aufgerufene Funktion erfolgreich war oder nicht, ist mit Sicherheit sinnvoll. Sollte es dennoch sinnvoll erscheinen, den Rückgabewert als irrelevant zu betrachten, dann muss dies kommentiert werden.

Regel 8:

Die Nutzung des Präprozessors muss auf die Inkludierung der Headerfiles sowie einfache Makrodefinitionen beschränkt werden. Komplexe Definitionen wie variable Argumentlisten, rekursive Makrodefinitionen usw. sind verboten. Bedingte Compilierung soll auf ein Minimum beschränkt sein.

Der Präprozessor kann (leider) so genutzt werden, dass er sehr zur Verwirrung von Softwareentwicklung und Codechecker beitragen kann, daher die Begrenzung. Die Anzahl der Versionen, die man mittels bedingter Compilierung und entsprechend vielen Compilerswitches erzeugen kann, wächst exponentiell: Bei 10 Compilerswitches erhält man bereits $2^{10} = 1024$ verschiedene Versionen, die alle getestet werden müssen.

Regel 9:

Die Nutzung von Pointer muss auf ein Minimum begrenzt sein. Grundsätzlich ist nur ein Level von Dereferenzierung zulässig. Pointer dürfen nicht durch Makros oder `typedef` verschleiert werden. Pointer zu Funktionen sind verboten.

Die Einschränkung bei Zeigern dürfte allgemein verständlich sein, insbesondere aber soll die Arbeit von Codecheckern nicht behindert werden.

Regel 10:

Der gesamte Code muss vom ersten Tag an so compiliert werden, dass die höchste Warnstufe mit allen Warnungen zugelassen eingeschaltet ist. Der Code muss ohne Warnungen compilieren. Der Code muss täglich gecheckt werden, möglichst mit mehr als einem Codeanalysator, und dies mit 0 Warnungen.

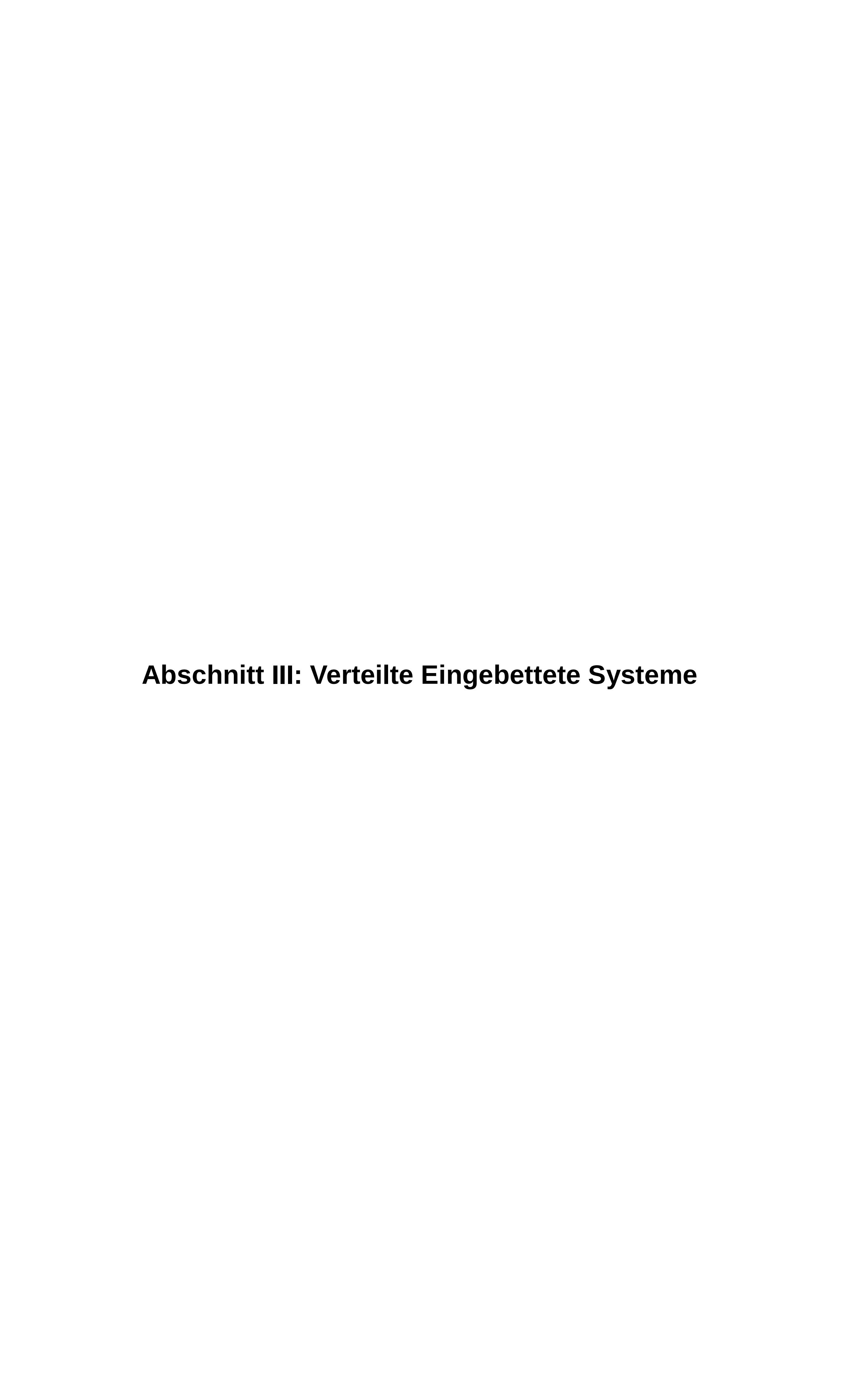
Diese Regel sollte peinlichst beachtet werden, denn Warnungen bedeuten immer etwas. Sollte die Warnung als verkehrt identifiziert werden, muss der Code umgeschrieben werden, denn dies kann auch bedeuten, dass der Codechecker den Teil nicht versteht.

Als Tipp für einen Codechecker: Lint bzw. splint (Secure Programming Lint) [lint].

7 Sichere Software und C

8 Hardwarenahe Programmierung

9 Hardware/Software Co-Design



Abschnitt III: Verteilte Eingebettete Systeme

10 Netzwerke und Standards

11 Design verteilter Applikationen im Bereich Eingebetteter Systeme



Abschnitt IV: Test und Verifikation

12 Softwaremetriken

13 Softwarequalität

Eingebettete Systeme sind immer Bestandteil einer übergeordneten Maschine; Fehler in der Software dieser Systeme können also zu Schädigungen der Maschine und von Menschen führen. Dies allein ist sicher Motivation genug, in die Softwarequalität zu investieren.

Dies ist eine hehre Aufgabenstellung, die schnell formuliert und schwierig umzusetzen ist. Zunächst werden Begriffe erläutert und Definitionen gegeben. Speziell auf das Thema Zuverlässigkeit zugeschnitten ist der nächste Abschnitt, gefolgt von einem Kapitel zum anderen Blickwinkel: Die Sicht der Maschine (bzw. Maschinenbauer). Den Abschluss bildet ein Vorschlag für Codierungsregeln in Projekten mit sicherheitskritischer Software.

13.1 Beispiele, Begriffe und Definitionen

13.1.1 Herausragende Beispiele

Leider gibt es einige herausragende, sehr bekannte Beispiele dafür, dass ein Software-basiertes System nicht ordnungsgemäß funktioniert hat. Hierzu zählen die Bruchlandung eines Airbus A-320 auf dem Warschauer Flughafen am 14.09.1993 und der Absturz der Ariane-5 am 04.06.1996 in Kourou, Französisch-Guayana.

Beim Beispiel der Bruchlandung des Airbus A-320 war die Ursache eine fehlerhafte Bodenberührungserkennung im Flugzeug. Bedingt durch plötzlich auftretenden, starken Seitenwind setzte der Airbus mit nur einem Rad auf dem Boden auf, die Software erkannte dies nicht als Bodenkontakt an und schaltete nicht aus dem Flight Mode heraus. Die Piloten konnten somit keine Schubumkehr einschalten, das Flugzeug kam nur wenig gebremst von der Landebahn ab, fing Feuer, so dass 2 Menschen starben und 54 verletzt wurden.

Der Fehler lag in der Entscheidung der Konstrukteure und Software-Ingenieure, wie die Messungen der Bodensensoren interpretiert wurden. Der aufgetretene Fall war nicht abgedeckt, und somit kam es zum Unglück.

Im zweiten Fall musste die europäische Trägerrakete Ariane 5 bei ihrem Jungfernflug gesprengt werden, weil sie von ihrer geplanten Bahn stark abwich und in bewohntes Gebiet abzustürzen drohte. Die Ursache hier war ein nicht abgefangener Datenüberlauf bei der Berechnung der Flugbahn. Die Software war einfach von der Vorgängerrakete übernommen worden, bei der bewiesen werden konnte, dass dieser Überlauf niemals stattfinden konnte. Die Ariane 5 hingegen war schubstärker, und die Rakete erreichte Geschwindigkeiten, deren interne Darstellung 32767 (16 bit Integer mit Vorzeichen) überschritt. Der Datenunterlauf führte dann

zur Bahnabweichung und zur Sprengung. Ein Klassiker unter den Softwarefehlern, der mithilfe von Datenbereichskontrollen hätte abgefangen werden können.

Beide Fehler resultierten in Tod, Verletzung oder Gefährdung von Menschen sowie in erhebliche wirtschaftliche Verluste, Kriterien dafür, dass die Systeme sicherheitskritisch waren.

13.2 Grundlegende Begriffe und Definitionen

Als zentral in einem modernen Projekt wird heute die Softwarequalität erachtet. Dabei stellt sich natürlich die Frage, was darunter eigentlich zu verstehen ist:

Definition 13.1 [ISO/IEC 9126]:

Softwarequalität ist die Gesamtheit der Merkmale und Merkmalswerte eines Softwareprodukts, die sich auf dessen Eignung beziehen, festgelegte oder vorausgesetzte Erfordernisse zu erfüllen.

Konkret wird die Beurteilung erst dann, wenn man sich auf die Qualitätsmerkmale bezieht. Diese stellen Eigenschaften einer Funktionseinheit dar, anhand deren ihre Qualität beschrieben und beurteilt werden. Allerdings enthalten sie keine Aussage über den Grad der Ausprägung. Beispielsweise existieren folgende **Softwarequalitätsmerkmale** (die im Übrigen miteinander in Wechselwirkung stehen oder voneinander abhängig sein können):

- Funktionalität
- Zuverlässigkeit
- Benutzbarkeit
- Effizienz
- Änderbarkeit
- Übertragbarkeit

Die nachfolgenden Definitionen stellen klar, was unter Softwarefehlern bzw. Fehlern allgemein verstanden wird. Hierbei wird zwischen tatsächlich auftretenden Fehlern, möglichen Fehlern und fehlerhaften Handlungen, die zu den beiden erstgenannten führen können, unterschieden:

Definition 13.2:

Failure (*Fehlverhalten, Fehlerwirkung, äußerer Fehler***)**: Hierbei handelt es sich um ein Fehlverhalten eines Programms, das während seiner Ausführung auch wirklich auftritt.

Definition 13.3:

Fault (*Fehler, Fehlerzustand, innerer Fehler*): Es handelt sich um eine fehlerhafte Stelle eines Programms, die ein Fehlverhalten auslösen kann.

Definition 13.4:

Error (*Irrtum, Fehlhandlung*): Es handelt sich um eine fehlerhafte Aktion, die zu einer fehlerhaften Programmstelle führt.

Daraus ergibt sich, dass Fehlhandlungen (*errors*) bei der Programmentwicklung oder durch äußere Einflüsse (z.B. Höhenstrahlung, Hardwareprobleme z.B. bei Flash-EEPROM-Zellen oder durch Bauteilestreuungen) zu Fehlern (*faults*) im Programm führen, die ihrerseits zu einem Fehlverhalten (*failure*) bei der Ausführung führen können. Hier soll die Qualitätssicherung entgegenwirken, und zwar sowohl konstruktiv als auch analytisch.

Um die Definitionen für *Validierung* und *Verifikation* zu verstehen, muss man den kompletten Designprozess betrachten (Bild 13.1). Aus einer informellen Problembeschreibung folgt eine formale Anforderungsdefinition, aus der heraus dann das eigentliche Rechnersystem (z.B. mit Mikroprozessor und Software) konstruiert wird. Die Übereinstimmung von Problem und Anforderungsbeschreibung ist sehr schwierig festzustellen, allein, weil die Problembeschreibung informell (und damit nicht maschinenprüfbar) ist. Dieser Vorgang wird Validierung genannt.

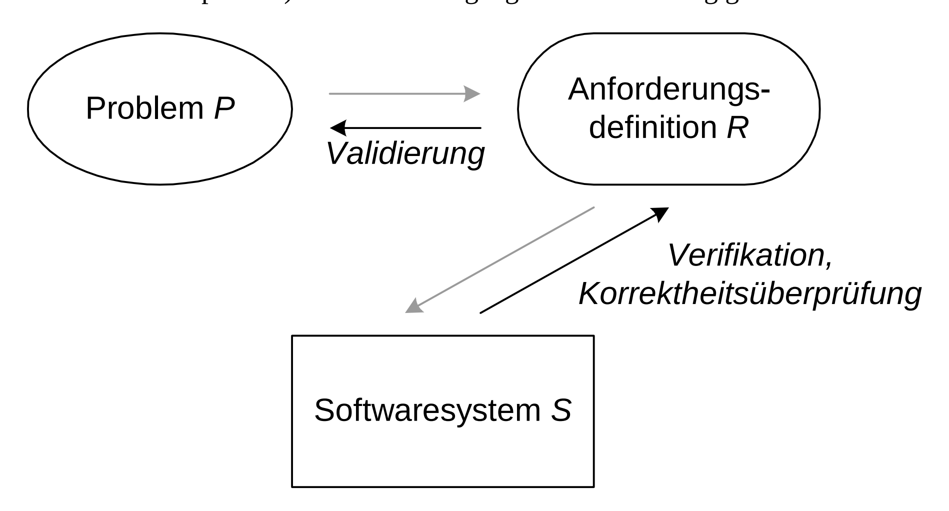


Bild 13.1 Einordnung der Begriffe Validierung und Verifikation

Die Verifikation hingegen in grundsätzlich durch formales Vorgehen lösbar, allerdings oft ebenfalls mit Schwierigkeiten. Hierzu sei einmal ein Software-basiertes System betrachtet: Eine logisch/arithmetische Anforderungsdefinition etwa in UML kann durch eine geeignete Software gegen ein daraus entstandenes Softwaresystem verifiziert werden (bzw. umgekehrt), mehr noch: Aus einer solchen Anforderungsdefinition kann mithilfe von Codegeneratoren das Softwaresystem sogar erzeugt werden.

Weitere Randbedingungen hingegen, wie sie z.B. in Form von zeitlichen Randbedingungen (Echtzeitsystem) vorliegen, können zwar formalisiert werden, sie sind jedoch meist nicht funktional (also durch einen Compiler übersetzbar) und im Zielsystem nicht (oder zumindest nur unter weiteren Randbedingungen) formal prüfbar. Hier spielt auch die Systemkonzeption eine große Rolle (→ 3, 4).

Die formale Verifikation ist damit nur ein Bestandteil der Maßnahmen zur Erhöhung der Softwarequalität, der weitaus größere besteht in dem Testen.

13.3 Zuverlässigkeit

Von elektronischen Systemen wird ein hohes Maß an Zuverlässigkeit erwartet. Dieser Satz kann sicherlich als allgemein gültig angesehen werden, aber was ist *Zuverlässigkeit* eigentlich?

Definition 13.5:

Zuverlässigkeit (*reliability*) ist die Wahrscheinlichkeit, dass ein System seine definierte Funktion innerhalb eines vorgegebenen Zeitraums und unter den erwarteten Arbeitsbedingungen voll erfüllt, das heißt intakt ist und es zu keinem Systemausfall kommt.

Definition 13.6:

Die *Verfügbarkeit* (*availability*) eines Systems ist der Zeitraum gemessen am Anteil der Gesamtbetriebszeit des Systems, in dem es für den beabsichtigten Zweck eingesetzt werden kann.

Defintion 13.7:

Ein *Systemausfall* (*failure*) liegt vor, wenn ein System sein geforderte Funktion nicht mehr erfüllt.

Definition 13.8:

Ein *Risiko* ist das Produkt der zu erwartenden Eintrittshäufigkeit (Wahrscheinlichkeit) eines zum Schaden führenden Ereignisses und des bei Eintritt des Ereignisses zu erwartenden Schadensausmaßes.

Mit *Grenzrisiko* wird das größte noch vertretbare Risiko bezeichnet.

Hier sollte ganz deutlich sein, dass das, was noch zumut- oder vertretbar ist, durch die technologische Machbarkeit beeinflusst (bzw. definiert) wird. Dies kann beispielsweise so geschehen, dass eine neue Maschine (z.B. Flugzeug) zugelassen bzw. zertifiziert wird, wenn eine katastrophale Fehlersituation nur noch mit einer Wahrscheinlichkeit von 10^{-9} pro Betriebsstunde auftreten kann, integriert über alle Maschinen dieses Typs. Wie dies berechnet werden kann steht u.a. in den Normen zur Maschinensicherheit (→ 14).

13.3.1 Konstruktive Maßnahmen

Eine der wichtigsten Fragen für die Konstruktion bzw. das Design sicherheitskritischer Maschinen ist diejenige nach konstruktiven Maßnahmen zur Vermeidung von Fehlern oder wenigstens Fehlerfolgen. Diese Art der Fehlertoleranz basiert immer auf einer Form der Redundanz, d.h. zur Erkennung von Fehlern sind mehr Informationen als zum eigentlichen Betrieb notwendig, daher wird das System komplexer.

Der naheliegende und vor einigen Jahren auch fast ausschließlich genutzte Ansatz liegt dabei in der Erweiterung der Hardware um fehlererkennende Teile wie Paritätsbits, Prüfsummen, fehlererkennende bzw. –korrigierende Codes usw. Dieser Ansatz wird aktuell jedoch als zu einengend angesehen, so dass man sich nun um Mischformen bemüht.

13.3.1.1 Einsatz redundanter Hardware

Redundante Hardware kann im wesentlichen durch Vervielfachung mit einem Mehrheitsentscheider erreicht werden. Dies wird auch als ”Voting“ bezeichnet, und bis auf den Entscheider selbst ist alles mehrfach ausgelegt.

Der Vorteil dieses Ansatzes liegt darin, dass die gleiche Hardware kopiert wird. Das Fehlermodell geht davon aus, dass die Hardware aufgrund eines Defektes nicht funktioniert, nicht aufgrund eines konstruktiven Mangels. Die eigentliche Fehlertoleranz, d.h., die fehlervermeidende Reaktion, kann dann in Form dreier Varianten erfolgen:

- **Statische Redundanz**: Die Hardware bleibt immer erhalten, die Mitglieder stimmen laufend (an vorgesehenen Punkten) ab, und die Mehrheitsentscheidung gilt.
- **Dynamische Redundanz**: Bei Erkennen eines Fehlers wird die fehlerhafte Hardware rekonfiguriert, d.h., Reservekomponenten kommen zum Einsatz. Hier existieren z.B. Modellen für Prozessoren, Operationen (wie Addition) auf andere Einheiten (bzw. eine Sequenz davon) abzubilden.
- **Hybride Ansätze**: Die Mischung aus Mehrheitsvotum und Rekonfiguration stellt einen hybriden Ansatz dar, der zwar komplexer ist, aber natürlich die größte Flexibilität besitzt.

Genau genommen darf man das Fehlermodell der Hardware, dass diese zunächst fehlerfrei ist und keinen konstruktiven Mangel hat, natürlich nicht unbedarft übernehmen. So sind so genannte Chargenprobleme bekannt, d.h., eine Produktionscharge eines Hardwarebausteins zeigt den gleichen Mangel. Dies würde zu einem übereinstimmenden Verhalten mehrerer Komponenten im Betrieb führen mit dem Ergebnis, dass die Fehlertoleranz in eine Fehlerakzeptanz übergeht.

Um solche Fälle auszuschließen müssen konstruktive Maßnahmen ergriffen werden, die dann verschiedene Hardwarekomponenten miteinander verbinden.

13.3.1.2 Einsatz redundanter Software

Der mehrmalige Einsatz der gleichen Software ist zwecks Fehlertoleranz sinnlos, da Software nicht altert und somit keine neuen Fehler entstehen. Fehler sind von Beginn an enthalten, um hier fehlertolerant zu sein, müssen verschiedene Versionen verwendet werden.

Dies bedeutet einfach, dass mehrere unabhängige Designteams verschiedene Versionen herstellen müssen. Auch hier kann dann wieder zwischen statischer und dynamischer Redundanz unterschieden werden:

- **Statische Redundanz** (N-Version-Programming): Es werden mehrere Versionen durch verschiedene Entwicklungsteams erstellt, die dann real oder im Zeitscheibenverfahren nebeneinander laufen. Und definierte Synchronisationspunkte haben. An diesen Synchronisationspunkten werden die Ergebnisse verglichen und durch einen Voter bestimmt, welches Ergebnis das wahrscheinlich richtige ist (Mehrheitsentscheidung). Diese Verfahren ist sehr aufwendig.
- **Dynamische Redundanz** (Recovery Blocks): Es wird eine permanente Fehlerüberwachung durchgeführt, um beim Erkennen eines Fehlers den entsprechenden Softwareblock gegen eine alternative Softwarekomponente auszutauschen.

13.3.2 Analytische Maßnahmen

Um bei komplexen Systemen die Zuverlässigkeit zu beurteilen muss man dieses in seine Einzelfunktionalitäten zerlegen. Die Zuverlässigkeit einer einzelnen Komponente sei dann bekannt und mit $R_i(t)$ mit $0 < R_i(t) < 1$ bezeichnet.

Die Kopplung der Systemkomponenten kann dann stochastisch abhängig oder unabhängig sein. Im einfacheren unabhängigen Fall müssen dann bei serieller Kopplung der Komponenten (heißt: das System fällt aus, wenn mindestens eine der Komponenten ausfällt) die Einzelwahrscheinlichkeiten multipliziert werden:

$$R_{seriell} = \prod_i R_i(t)$$

Bei paralleler Kopplung – in diesem Fall soll das System noch intakt sein, wenn mindestens eine Komponente intakt ist – ergibt sich die Zuverlässigkeit

$$R_{parallel} = 1 - \prod_i [1 - R_i(t)]$$

Bei stochastischer Abhängigkeit wird die Analyse entschieden komplexer, denn hier bewirken Einzelausfälle Kopplungen zu anderen. In diesem Fall kommen Analyseverfahren wie z.B. Markovketten zum Einsatz.

13.3.3 Gefahrenanalyse

Unter Gefahrenanalyse wird ein systematisches Suchverfahren verstanden, um Zusammenhänge zwischen Komponentenfehlern und Fehlfunktion des Gesamtsystems aufzudecken. Hierzu müssen noch einige Begriffe definiert werden:

Definition 13.9:

Als **Gefahr** (hazard) wird eine Sachlage, Situation oder Systemzustand bezeichnet, in der/dem eine Schädigung der Umgebung (Umwelt, Maschine, Mensch) möglich ist.

Ein Gefahrensituation ist also eine Situation, in der das Risiko größer als das Grenzrisiko ist. Die ursächlich zugrundeliegenden Fehler sollen nun zurückverfolgt werden, unabhängig davon, ob diese zufällig (Alterung) oder konstruktiv bedingt sind.

Definition 13.10:

Tritt eine Schädigung tatsächlich ein, so bezeichnet man dieses Ereignis als **Unfall** (accident).

Die systematischen Suchverfahren können nun prinzipiell überall ansetzen, in der Praxis wählt man jedoch einen der beiden Endpunkte. Man spricht dann von Vorwärts- bzw. Rückwärtsanalyse. Bekannt sind hierbei die Ereignisbaumanalyse (FTO, *Fault Tree Analysis*) und die *Failure Mode and Effect Analysis* (FMEA). Im letzteren Fall werden folgende Fragestellungen untersucht:

- Welche Fehler(-ursachen) können auftreten?
- Welche Folgen haben diese Fehler?
- Wie können diese Fehler vermieden oder das Risiko minimiert werden?

Die Fehlerliste führt dann zu einer Systemüberarbeitung, und die Analyse beginnt von vorne. Die FMEA hat folgende Ziele:

- Kein Fehler darf einen negativen Einfluss (auf redundante Systemteile) haben.
- Kein Fehler darf die Abschaltung der Stromversorgung eines defekten Systemteils verhindern.
- Kein Fehler darf in kritischen Echtzeitfunktionen auftreten.

Letztendlich ist dies auch Forschungsthema. So gibt es in Deutschland beispielsweise die Initiative ”Organic Computing“, die Methoden der Biologie nachzuvollziehen versucht.

13.3.4 Software-Review und statische Codechecker

Software Review ist ein Teil des analytischen Prozesses, der alleine aufgrund der Trefferquote zwingend notwendig ist: 30 – 70 % aller Fehler werden in dieser Phase gefunden. Leider kostet ein solches Review, wird es ernsthaft betrieben, sehr viel Zeit.

Eine gewisse Hilfe sind die statischen Codechecker, die den Code analysieren und wertvolle Hinweise liefern. In [lint] kann z.B. ein von *lint* abstammender statischer Codechecker als Freeware-Tool gefunden werden.

Statische Codechecker können z.B. folgende Aktionen durchführen:

- Initialisation Tracking: Variablen werden darauf untersucht, ob sie vor der ersten lesenden Verwendung initialisiert wurden. Dies erfolgt auch über if/else-Konstrukte usw., so dass – im Gegensatz zu vielen Compilern – wirkliche Initialisierungsfehler gefunden werden.
- Value Tracking: Indexvariable für Arrays, mögliche Divisionen durch Null sowie Null-Zeiger stellen potenzielle Fehlerquellen im Programm dar. Sie werden ausführlich analysiert.
- Starke Typprüfung: Abgeleitete Typen (#typedef in C) werden darauf überprüft, dass nur sie miteinander verknüpft werden (und nicht die Basistypen). Weiterhin erfolgt eine sehr genaue Typprüfung, also z.B., ob Vergleiche zwischen int und short usw. geführt werden, und eine entsprechende Warnung wird ausgegeben.
- Falls es so genannten Funktionssemantiken gibt – das sind Regeln für Parameter und Rückgabewerte, etwa so, dass der erste Funktionsparameter nicht 0 sein darf – dann sind weitere Checks möglich.

Letztendlich erzwingt der Einsatz von statischen Codecheckern, dass sich der Entwickler sehr um seinen Sourcecode bemüht. Und genau das dürfte in Zusammenhang mit Codierungsregeln (→ 6.6) einen sehr positiven Effekt auf die Softwarequalität haben

13.3.5 Testen (allgemein)

In der Praxis steuert alles auf das Testen hin, dies erscheint als die ultimative Lösung. Ein gute Einführung in dieses überaus komplexe Thema ist in [Grü04a] [Grü04b] [Grü05a] [Grü05b] und [Grü06] gegeben.

Testen muss als destruktiver Prozess verstanden werden. Man versucht, die Software zu brechen, ihre Schwachpunkte zu finden, Fehler aufzudecken. Es ist natürlich sehr schwierig für den Entwickler, sein bislang konstruktive Sicht aufzugeben: Bislang war er/sie während des Designs und des Programmierens damit befasst, eine ordentliche Software herzustellen, so dass die destruktive Sicht sicherlich schwer fallen würde. Aus diesem Grund muss der Test von anderen, nicht mit der Entwicklung befassten Personen durchgeführt werden.

Um den Testprozess genauer zu beschreiben, wird er in 4 Phasen eingeteilt [Grü04a]:

- Modellierung der Software-Umgebung
- Erstellen von Testfällen

- Ausführen und Evaluieren der Tests
- Messen des Testfortschritts

13.3.5.1 Modellierung der Software-Umgebung

Eine der wesentlichen Aufgaben des Testers ist es, die Interaktion der Software mit der Umgebung zu prüfen und dabei diese Umgebung zu simulieren. Dies kann eine sehr umfangreiche Aufgabe sein:

- Die klassische Mensch/Maschine-Schnittstelle: Tastatur, Bildschirm, Maus. Hier gilt es z.B., alle erwarteten und unerwarteten Eingaben und Bildschirminhalte in dem Test zu organisieren. Einer der Ansätze hierzu heißt Replay-Tools, die Eingaben simulieren und Bildschirminhalte mit gespeicherten Bitmaps vergleichen können.
- Das Testen der Schnittstelle zur Hardware: Ideal ist natürlich ein Test in der Form ”Hardware in the loop“, d.h., die zu testende Hardware ist vorhanden und offen. Falls nicht, müssen hier entsprechende Umgebungen ggf. sogar entwickelt werden. Zudem gilt es, bei dem Test auch nicht-erlaubte Fälle einzubinden, d.h., es müssen Fehler in der Hardware erzeugt werden, insbesondere bei Schnittstellen.
- Die Schnittstelle zum Betriebssystem ist genau dann von Interesse, wenn Dienste hiervon in Anspruch genommen werden. Hier sind Fehlerfälle, z.B. in Form zu geringen Speicherplatzes auf einem Speichermedium oder Zugriffsfehlern, zu testen.
- Dateisystem-Schnittstellen gehören im Wesentlichen auch zum Betriebssystem, seien hier jedoch explizit erwähnt. Der Tester muss Dateien mit erlaubtem und unerlaubtem Inhalt sowie Format bereitstellen.

Letztendlich ist es der Phantasie und der Erfahrung des Testers zu verdanken, ob ein Test möglichst umfassend oder eben ein ”Schönwettertest“ ist. Beispielsweise müssen oft ungewöhnliche Situationen getestet werden, wie z.B. der Neustart einer Hardware während der Kommunikation mit externen Geräten.

13.3.5.2 Erstellen von Testfällen

Das wirkliche Problem der Erstellung von Testfällen ist die Einschränkung auf eine handhabbare Anzahl von Test-Szenarien. Hierbei hilft (zumindest ein bisschen) die so genannte *Test Coverage*: Man stellt sich die Frage, welche Teile des Codes noch ungetestet sind. Hierfür sind Tools erhältlich (bzw. in Debugging-Tools eingebaut), die den Sourcecode anhand der Ausführung kennzeichnet. Mit dem Ziel, die gewünschte Testabdeckung am Quellcode zu erreichen, wird der Tester daher Szenarien auswählen, die

- typisch auch für die Feldanwendung sind;

- möglichst ”bösartig“ sind und damit eher Fehler provozieren als die bereits zitierten ”Schönwettertests“;
- Grenzfälle ausprobieren

Bei der Testabdeckung gilt es noch zu überlegen, ob die Ausführung einer Sourcecodezeile überhaupt genügt. Hierzu werden noch Testabdeckungsmetriken dargestellt (→ 13.3.7)

13.3.5.3 Ausführen und Evaluieren der Tests

Zwei Faktoren beeinflussen die Ausführung des Tests, der manuell, halbautomatisch oder vollautomatisch sein kann: die Haftung bei Software-Fehlern und die Wiederholungsrate der Tests. Anwendungen mit Sicherheitsrelevanz etwa erzeugen einen erheblichen Druck in Richtung automatischer Tests, allein, um die exakte Wiederholbarkeit zu erreichen.

Derartige Wiederholungen können notwendig sein, wenn an anderer Stelle ein Fehler gefunden wurde, dessen Behebung nun auf Rückwirkungsfreiheit getestet werden soll (so genannte Regressionstests).

Nach Ausführung der Tests, was sehr gut automatisch durchführbar ist, müssen die Tests bewertet werden, was meist nicht automatisch durchzuführen ist. Zumindest müssen die Kriterien, wann ein Test bestanden ist und wann nicht, vorher fixiert werde, ansonsten droht ein pures ”Herumprobieren“. Last not least bleibt die Frage der Vertrauenswürdigkeit des Tests, denn ein ständiger Erfolg sollte Misstrauen erzeugen. Um dies zu prüfen, werden bewusst Fehler eingebaut (Fault Insertion oder Fault Seeding), deren Nichtentdeckung natürlich eine Alarmstufe Rot ergäbe.

13.3.5.4 Messen des Testfortschritts

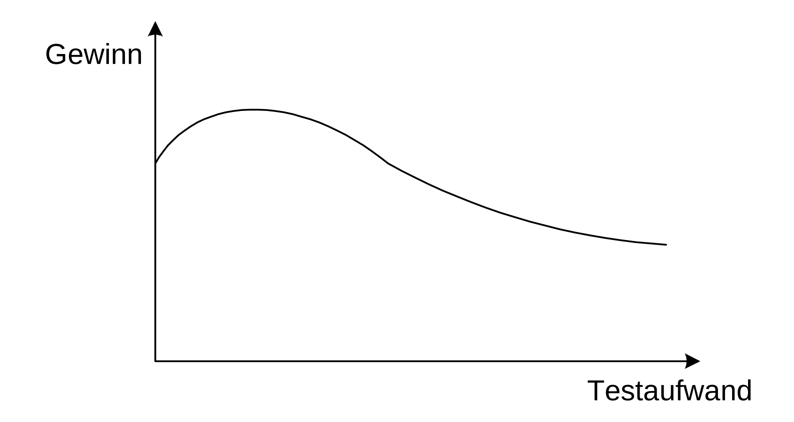


Bild 13.2 Gewinn versus Testaufwand

Ein Testprojekt sollte wie jedes andere Projekt genau geplant werden. Teil dieses Plans ist die Festlegung des Projektziels, etwa in der Form, wie viele unentdeckte Fehler die Software nach Testende noch haben darf. Art und Umfang der Tests werden sich nach dieser Größe richten, insbesondere darf nicht übersehen werden, dass sich der differenzielle Gewinn mit wachsendem Testaufwand wieder erniedrigt (Bild 13.2).

Um dies zumindest abschätzen zu können, ist das Wissen über die Komplexität des Codes wichtig. Eine passende Codemetrik ist die zyklomatische Komplexität (cyclomatic complexity) nach McCabe: Diese bestimmt die Anzahl der if-, while-do- und for-Kommandos im Code und damit die Anzahl der möglichen Verzweigungen. Tools hierfür sind (auch frei) verfügbar.

13.3.6 Modultests

Die meisten Software-Entwicklungsmodelle unterscheiden zwischen Modultests, Integrationstests und Systemtests. Modultests sind dabei das erste und wirkungsvollste Instrument, denn durchschnittlich 65% aller nicht schon in Reviews abgefangener Software-Fehler werden hier gefunden.

Für einen Modultest kann man verschiedene Strategien anwenden. Ein möglicher Weg kann der folgende sein:

1. Man teilt alle Eingangsgrößen (Variablen) in so genannte Äquivalenzklassen ein. Eine Äquivalenzklasse enthält all jene Eingangsgrößen oder Resultate eines Moduls, für die erwartet wird, dass ein Programmfehler entweder alle oder keinen Wert betrifft.

   Beispiel: Die Absolut-Funktion int abs(int) besitzt drei Äquivalenzklassen: negative Werte, die Null und positive Werte.

2. Aus jeder Äquivalenzklasse nimmt man nun zum Test des Moduls mindestens einen Vertreter. Im Testdesign werden die Eingangswerte, die Aktion und die erwarteten Ergebnisse festgelegt. Bei der Testdurchführung werden dann die erwarteten mit den tatsächlichen Ergebnisse verglichen, wobei ggf. ein Toleranzbereich zu definieren ist (z.B. bei Floating-Point-Zahlen).

Dieser Test orientiert sich nicht am inneren Design des Moduls und wird daher auch als ”Black-Box-Test“ bezeichnet. Wichtig ist dabei auch die Erkenntnis, dass ggf. auch Software zum Testen geschrieben werden muss, z.B. zum Aufruf, oder falls auf andere, noch nicht fertige oder nicht getestete Module zurückgegriffen wird. Im letzteren Fall werden die fehlenden Module durch so genannte Programmstümpfe (program stubs) ersetzt.

Der Test wird im Allgemeinen ergeben, dass keineswegs alle Codezeilen durchlaufen wurden. Um dies auch wirklich nachweisen zu können, werden Test-Coverage-Tools eingesetzt. Diese instrumentieren den Originalcode, d.h., sie fügen Code hinzu, der dem Tool den Durchlauf meldet. Nach dieser ersten Testphase werden also weitere Schritte folgen:

3. Der bisherige Test wird analysiert, und die Test Coverage wird bestimmt. Hieraus soll der Tester nun ableiten, mithilfe welcher Einganswerte er weitere Teile durchlaufen und damit testen lassen kann. Der Test wird dann mit den neuen Werten weitergeführt, bis eine zufriedenstellende Test Coverage erreicht ist.

Diese Form des Tests wird "White-Box-Test“ genannt, da nun die Eigenschaften des Quellcodes ausgenutzt werden.

Weiterhin entsteht die Frage nach dem Testsystem: Host- oder Target-Testing? Grundsätzlich heißt die Antwort natürlich Zielsystem, denn nur hier können versteckte Fehler wie Bibliotheksprobleme, Datentypabweichungen (wie viele Bits hat int?) usw. erkannt werden. Weiterhin können gemischte C/Assemblerprogramme tatsächlich nur dort getestet werden.

In der Praxis weicht man jedoch häufig auf Hostsysteme aus, weil diese besser verfügbar sind, Festplatte und Bildschirm haben, ggf. schneller sind usw.

13.3.7 Integrationstests

Der Test der einzelnen Module erscheint vergleichsweise einfach, da insbesondere die Modulkomplexität in der Regel noch begrenzt sein wird. Der nun folgende *Integrationstest* fasst nun mehrere (bis alle) Module zusammen, testet die Schnittstellen zwischen den Modulen und ergibt hiermit den Abschlusstest der Software, da der darauf folgende Systemtest auf das gesamte System einschließlich Hardware zielt.

13.3.7.1 Bottom Up Unit Tests

Die wohl sicherste Integrationsteststrategie besteht darin, keinen expliziten Integrationstest zu machen und stattdessen die Modultests entsprechend zu arrangieren. Dies wird als *Bottom Up Unit Test* (BUUT) bezeichnet.

Wie beim Black-Box-Modultest, auch als Isolationstest bezeichnet, werden die low-level-Module einzeln getestet, indem sie von einer Testumgebung (stubs, drivers) umfasst werden. Sind diese Module hinreichend getestet, werden sie zu größeren Modulen zusammengefasst und erneut getestet, wobei "höhere“ Softwaremodule nur auf bereits getestete Module zurückgreifen dürfen.

Der Ansatz hört sich gut an, ist auch wirklich die sauberste Methode, hat aber auch Nachteile:

- Die Entwicklung wird erheblich verlangsamt, da Entwicklung und Test sozusagen Hand in Hand gehen müssen. Zudem ist eine erhebliche Menge an Code zusätzlich zu schreiben (stubs, driver).
- Folglich wird sich die BUUT-Methode auf kleinere Softwareprojekte beschränken.

- Das Softwareprojekt muss von Beginn an sehr sauber definiert sein, d.h., die Modulhierarchie muss streng gewährleistet sein.

13.3.7.2 Testabdeckung der Aufrufe von Unterprogrammen

Die zweite Methode zum Integrationstest besteht in einer möglichst hohen Abdeckung aller Unterprogrammaufrufe (call pair coverage). Messtechnisch wird der Code hierzu wiederum instrumentiert, d.h. mit zusätzlichem Code zur Messung der Abdeckung versehen. Es wird nun verlangt, eine 100% Call Pair Coverage zu erreichen.

Wird diese Abdeckung nicht erreicht, bedeutet dies, dass die erdachten Fälle zum Integrationstest nicht die volle Systemfunktionalität abdecken, und es muss nachgebessert werden.

13.3.7.3 Strukturiertes Testen

Die *strukturierten Integrationstests* (SIT) wurden 1982 von Thomas McCabe eingeführt. Sie beruhen darauf, die minimal notwendige Anzahl von voneinander unabhängigen Programmpfaden zu bestimmen. Unabhängig ist dabei ein Programmpfad, wenn er nicht durch eine Linearkombination anderer Programmpfade darstellbar ist.

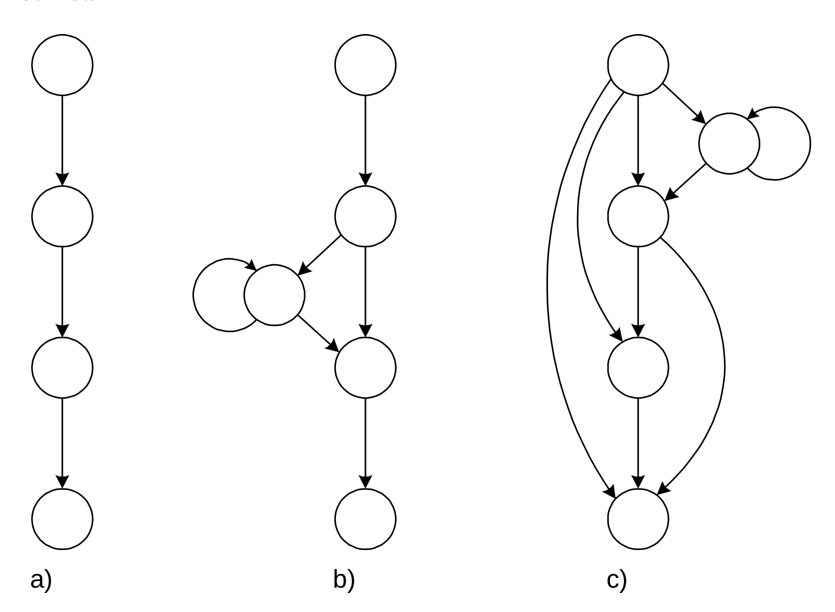


Bild 13.3 Kontrollflussgraphen mit den zyklomatischen Komplexitäten a) 1 b) 3 c) 6

Ausgangspunkt ist dabei ein Kontrollflussgraph des Programms (Bild 13.3). Hierin werden die voneinander unabhängigen Programmpfade bestimmt, dies ergibt die so genannte zyklomatische Komplexität. Es gilt hier die Formel

$$CC = E - N + 2$$

$$mit\ E = Anzahl\ der\ Kanten, N = Anzahl\ der\ Knoten$$

Für den Integrationstest kann der Graph reduziert werden, denn hier sollen ja nur die Aufrufe der Unterprogramme getestet werden. Alle Programmpfade, die keinen solchen Aufruf enthalten, können somit ausgeschlossen werden, allerdings nur unter der Voraussetzung, dass das Dateninterface zu den Unterprogrammen ausschließlich über Parameter realisiert ist. In diesem Fall können folgende Operationen zur Reduktion durchgeführt werden:

1. Alle Knoten, die ein Unterprogramm aufrufen, werden markiert.
2. Alle markierten Knoten dürfen nicht entfernt werden.
3. Alle nicht markierte Knoten, die keine Verzweigung enthalten, werden entfernt.
4. Kanten, die zum Beginn einer Schleife führen, die nur unmarkierte Knoten enthält, werden entfernt.
5. Kanten, die zwei Knoten so verbinden, dass kein Alternativpfad für diese Verbindung mit markierten Knoten existiert, werden entfernt.

Der reduzierte Graph muss nun nur noch getestet werden.

13.3.8 Systemtests

Zum Schluss folgen die *Systemtests*: Sie beziehen sich auf das gesamte System, also die Zusammenfügung von Hard- und Software. Hierbei ist häufig Kreativität gefordert, denn dem Test fehlt ggf. die Außenumgebung.

Einige Möglichkeiten, wie Teiltests aussehen können, seien hier aufgezählt:

- **Belastungs- und Performancetests**: Diese stellen fest, wie das Verhalten unter erwarteter Last (Performancetest) bzw. unter Überlast (Belastungstest) ist. Was hierbei eine Überlast ist, ist wiederum nicht exakt definierbar, aber es gibt Anhaltspunkte. So können Eingaberaten höher sein als die Pollingrate bei Timer-triggered- bzw. Event-triggered-Systemen, Geräte, die das System beeinflussen, werden auf höchste oder niedrigste Geschwindigkeit gestellt usw.
- **Failover und Recovery Test**: Hier wird geprüft, wie sich verschiedene Hardwareausfälle bemerkbar machen, ob beispielsweise Daten verloren gehen, inkonsistente Zustände erreicht werden usw.
- **Ressource Test**: Die im Vordergrund stehende Frage ist hier, ob die Hardwareressourcen ausreichen. Beispiel ist hier der Hauptspeicher, wobei Stack und Heap spezielle Kandidaten sind, denn deren Verhalten ist zumeist unberechenbar. Bei beiden gilt: Großzügige Dimensionierung schafft Vertrauen.

- **Installationstests**: Installationstests verfolgen zwei Ziele: Die Installation der Software muss unter normalen wie abnormalen (zu wenig Speicher, zu wenig Rechte usw.) Bedingungen korrekt verlaufen, und die Software muss danach auch richtig lauffähig sein. Letzteres muss vor allem dann getestet werden, wenn es bereits eine Installation gab.
- **Security Testing**: Dieser Test betrifft die Sicherheit, d.h., inwieweit das System vor Hackern oder anderen Angreifer geschützt ist. Hierzu muss sich der Entwickler so verhalten wie ein Hacker und versuchen, in das System einzudringen.

13.4 Die andere Sicht: Maschinensicherheit

Letztendlich ist entscheidend, was die Anwender von Software-basierten Systemen haben wollen bzw. welche Eigenschaften sie garantiert haben wollen. Die Funktionalität einschließlich der Zuverlässigkeit ist nämlich entscheidend für die Sicherheit der Maschinen, in die diese Systeme eingebaut sind.

Die entscheidenden neuen Normen zur Maschinensicherheit sind DIN ISO 13849 (Maschinensicherheit, voraussichtlich 2006 gültig) und DIN EN 61508 (Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme, Oktober 2005). Diese beiden sind eng aufeinander bezogen und verweisen gegenseitig. Tabelle 13.1 zeigt die so genannten Performance Level (PL) bzw. Security Integrity Level (SIL), die in den jeweiligen Normen definiert werden.

Wahrscheinlichkeit eines gefahrb. Ausfalls pro Stunde [1/h]	**PL,–ISO 13849-1**	**SIL,–EN IEC 61508**
10^{-5} <PDF< 10^{-4}	a	
$3x10^{-6}$ <PDF< 10^{-5}	b	1
10^{-6} <PDF< $3x10^{-6}$	c	1
10^{-7} <PDF< 10^{-6}	d	2
10^{-8} <PDF< 10^{-7}	e	3

Tabelle 13.1 Vergleich PL und SIL (PDF: Probability of dangerous failures per hour, auch PFH abgekürzt)

Interessant ist dabei die Sicht auf elektronische bzw. programmierbare elektronische Systeme. Programmierbare Hardware gilt dabei als Hardware. Wenn man nun ein sicheres System aufbauen will, müssen zusätzlich zu allen anderen Fehlern auch die Common Causation Failure (CCF), also die Fehler gleichen Ursprungs, beachtet werden.

Normalerweise reicht eine einfache Redundanz, also die Verdopplung der Hardware mit einer Entscheidungsinstanz aus, wenn es einen sicheren Zustand gibt.

Hiermit ist gemeint, dass dieser sichere Zustand angenommen wird, wenn eine Hardware (Überwachung) eine entsprechende Situation detektiert. Die CCF entstehen nun durch Bausteinfehler, die gemeinsam in beiden Bausteinen sind. Die Maschinensicherheit fordert daher bei sicherheitskritischen Applikationen eine ”diversitäre Redundanz“, d.h. zwei verschiedene Bausteine mit zwei verschiedenen Konfigurationen (falls es sich um programmierbare Hardware handelt).

Die Software in derartigen Systemen muss entweder redundant diversitär aufgebaut sein – dies bedeutet, dass unterschiedliche Compiler eingesetzt und zwei verschiedene Versionen von unterschiedlichen Designteams erstellt werden müssen –, oder die Software muss in einem komplexen Prozess zertifiziert werden – oder auch beides.

14 Test und Testmetriken

Literatur

[ASU99] *Alfred V. Aho, Rave Sethi, Jeffrey D. Ullman*, ”Compilerbau 1“. 2. Auflage, Oldenbourg Verlag, München, 1999.

[BBM00] *Benini, L.; Bogliolo, A.; De Micheli, G.*: A Survey of Design Techniques for System-Level Dynamic Power Management. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 8, No. 3, pp. 299–316 (2000).

[BH04] *Beierlein, T.; Hagenbruch, O. (Hrsg.)*: ”Taschenbuch Mikroprozessortechnik”. Fachbuchverlag Leipzig im Carl Hanser Verlag, München Wien, 3., aktualisierte und erweiterte Auflage, April 2004. ISBN 3-446-22072-0

[Bro+00] *Brooks, D. et.al.*: Power-Aware Microarchitecture. IEEE Micro Vol. 20, No. 6, pp. 26–44 (2000).

[CKURS] http://www.hmh-ev.de/files/ckurs.pdf

[Dea04] *Alexander G. Dean*, ”Efficient Real-Time Fine-Grained Concurrency on Low-Cost Microcontrollers”. IEEE Micro 24(4), pp. 10-22 (2004).

[GM03] *Timo Gramann, Dirk S. Mohl*, ”Precision Time Protocol IEEE 1588 in der Praxis”. Elektronik 52(24), S. 86–94 (2003).

[Grü04a] Stephan Grünfelder, ”Den Fehlern auf der Spur. Teil 1: Das Handwerk des Testens will gelernt sein, wird aber kaum gelehrt“. Elektronik 53(22) S. 60 .. 72 (2004).

[Grü04b] Stephan Grünfelder, Neil Langmead ”Den Fehlern auf der Spur. Teil 2: Modultests: Isolationstests, Testdesign und die Frage der Testumgebung“. Elektronik 53(23) S. 66 .. 74 (2004).

[Grü05a] Stephan Grünfelder, ”Den Fehlern auf der Spur. Teil 3: Automatische statische Codeanalyse“. Elektronik 54(9) S. 48 .. 53 (2005).

[Grü05b] Stephan Grünfelder, ”Den Fehlern auf der Spur. Teil 4: Integrationstests – das ungeliebte Stiefkind“. Elektronik 54(13) S. 73 .. 77 (2005).

[Grü06] Stephan Grünfelder, ”Den Fehlern auf der Spur. Teil 5: Systemtests – die letzte Teststufe ist alles andere als eine exakte Wissenschaft“. Elektronik 55(14) S. 45 .. 51 (2006).

[Hat95] *Les Hatton*, ”Safer C: Developing Software for High Integrity and Safety Critical Systems“. McGraw-Hill Professional, 1995.

[Hol06] Gerad J. Holzmann, ”The Power of 10: Rules for Developing Safety-Critical Code“. IEEE Computer 39(6), pp. 95–97, 2006.

[IEE1588] http://ieee1588.nist.gov/

[Lance2] http://www.lancecompiler.com/

[Laplace] Laplace-Operator: http://www.iiit.uni-karlsruhe.de/download/Versuch8_Bildverarbeitung_06.pdf

[lint] http://www.splint.org

[MP01] *Mudge, T.*: Power: A First-Class Architectural Design Constraint. IEEE Computer Vol. 34, No. 4, pp. 52–58 (2001).

[Sch05] *Scholz, P.*: Softwareentwicklung eingebetteter Systeme. Springer Verlag Berlin Heidelberg New York, 2005.

[Sie04] *Christian Siemers*, "Prozessor-Technologie". tecCHANNEL-Compact, Verlag Interactive GmbH, München, Mai 2004.

[Sie07a] *Christian Siemers*, "Hochsprachenprogrammierung: Einblick in die Arbeitsweise von Compilern – Teil 1". ElektronikPraxis 43, Sonderheft 1/2007 Embedded Systems, S. 46-47 (2007)

[Sie07b] Christian Siemers, "Hochsprachenprogrammierung: Einblick in die Arbeitsweise von Compilern – Teil 2". Embedded Software Engineering Report 2(2), S. 6, und http://www.ese-report.de/ (2007)

[Sie07c] Christian Siemers, " Hochsprachenprogrammierung: Einblick in die Arbeitsweise von Compilern – Teil 3". Embedded Software Engineering Report 2(3), S. 3, und http://www.ese-report.de/ (2007)

[SWM01] *Steinke, S.; Wehmeyer, L.; Marwedel, P.*: Software mit eingebautem Power-Management. Elektronik Vol. 50, H. 13, S. 62–67 (2001).

[SWW99] *Schmitt, F.-J.; von Wendorff, W.C.; Westerholz, K.*: Embedded-Control-Architekturen. Carl Hanser Verlag München Wien, 1999.

[TM05] *Emil Talpes, Diana Marculescu*, "Toward a Multiple Clock/Voltage Island Design Style for Power-Aware Processors". IEEE Transactions on Very Large Scale Integration (VLSI) Systems 13(5), pp. 591 – 603 (2005).

[4] *Falk, H.; Marwedel, P.*: Source Code Optimization Techniques for Data Flow Dominated Embedded Software. Kluwer Academic Publishers Boston Dordrecht London, 2004.

[5] *Booch, G.; Rumbaugh, J.; Jacobson, I.*: Das UML-Benutzerhandbuch. Addison-Wesley München, 2. Auflage, 1999.

[6] http://www.systemc.org

[11] *Uwe Schneider, Dieter Werner* (Hrsg.): ”Taschenbuch der Informatik“. Fachbuchverlag Leipzig im Carl Hanser Verlag, 3. Auflage,. München Wien 2000.

[12] http://www.faqs.org/faqs/realtime-computing/list/

[13] http://www.dedicated-systems.com/encyc/buyersguide/rtos/Dir228.html

[14] ”Special Issue on Real-Time Systems”. Proceedings of the IEEE 82(1), January 1994.

[15] ”Special Issue on Real-Time Systems”. Proceedings of the IEEE 91(7), July 2003.

[18] *Holger Blume, Hendrik T. Feldkämper, Thorsten von Sydow, Tobias G. Noll,* ”Auf die Mischung kommt es an - Probleme beim Entwurf von zukünftigen Systems on Chip“. Elektronik 53(19) S. 54 – 59, und Elektronik 53(20) S. 62 – 67 (2004).

[21] http://www.esterel-technologies.com/

Sachwortverzeichnis

D

E

T

U

V

W

Z